ENABLING EFFICIENT GRAPH COMPUTING WITH NEAR-DATA PROCESSING TECHNIQUES

A Dissertation
Presented to
The Academic Faculty

By

Lifeng Nai

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2017

ENABLING EFFICIENT GRAPH COMPUTING WITH NEAR-DATA PROCESSING TECHNIQUES

Approved by:

Dr. Hyesoon Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Moinuddin K. Qureshi
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sung Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computer Science
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Date Approved: December 15, 2016

To my family.

ACKNOWLEDGEMENTS

The completion of a Ph.D. dissertation is a long and rough journey, which I could never have done alone. I would like to take this opportunity to thank many people that have helped and supported me along the way.

First of all, I would like to express my sincere gratitude to my advisor, Dr. Hyesoon Kim, for providing the support when it is needed the most, for guiding my research with constructive and insightful advices, and for motivating me and many others with her great research attitude. I am extremely fortunate to have such a wonderful advisor, who provides not only the flexibility of pursuing research topics that I am interested in, but also the assistance and guidance in every aspect of a research project, from identifying research topics and polishing ideas to writing papers and even performing experiments. Without her advisement, I could never have finished this dissertation.

I would like to thank my former advisor, Dr. Hsien-Hsin S. Lee and co-advisor, Dr. Bo Hong for introducing me into the field of computer architecture and for their assistance and guidance during my initial years at Georgia Tech. I would also like to thank Dr. Moinuddin K. Qureshi, Dr. Sung Kyu Lim, Dr. Richard Vuduc, and Dr. Santosh Pande for being members of my dissertation committee and for providing valuable comments and suggestions.

I would like to thank Dr. Yinglong Xia, Dr. Ilie G Tanase, Dr. Ching-Yung Lin, Dr. Chen-Yong Cher, Dr. Jui-Hsin Lai, Jason Crawford, Dr. Toyotaro Suzumara, Dr. Jie Lu, and Dr. Chun-Fu Chen for their mentorship and friendship at IBM Research. They have been fantastic mentors and friends. I am fortunate to have the opportunity of working with them during the internships at IBM. A special thanks to Dr. Yinglong Xia for introducing me into the field of graph computing, which eventually becomes my dissertation topic. His insights and creativity have a profound influence on my graduate study.

My thanks to current and former HPArch members for their research collaborations and

for their friendship: Dr. Jen-Cheng Huang, Dr. Jaewoong Sim, Ramyad Hadidi, Hyojong Kim, Dr. Joo Hwan Lee, Pranith Kumar, Dilan Manatunga, and Prasun Gera. I also thank the former members of MARS: Dr. Sungkap Yeo, Dr. Mohammad M. Hossain, Guanhao Shen, Tzu-Wei Lin, Eric Fontaine, Andrei Bersatti, Dr. Nak Hee Seong, and Dr. Dean Lewis. A special thanks to Dr. Jen-Cheng Huang, Dr. Sungkap Yeo, and Dr. Mohammad M. Hossain for their incredible help and encouragement during the most difficult time of my Ph.D. study. And I am also grateful for having the opportunity of collaborating with Dr. Jen-Cheng Huang, Dr. Jaewoong Sim, Ramyad Hadidi, Hyojong Kim, and Pranith Kumar in multiple exciting research projects.

I thank my friends who have helped me during these years at Atlanta: Kehuang Li, Jian Huang, He Xiao, Xianzhi Meng, Qining Sun, Lingchen Zhu, Guanhua Feng, Xinwei Chen, and You-Chi Cheng for their friendship and companionship. Kehuang Li and I have been old friends since undergraduate school, and I am grateful for his support and friendship. Jian Huang and I always have discussions from time to time on various research topics, and I am really thankful to his great research insight and valuable suggestions. I thank He Xiao for the nice coffee breaks and wonderful research discussions together. I also thank Xianzhi Meng, Qining Sun, and Lingchen Zhu for being excellent roommates and for their help of numerous times in the daily life.

Above all, I am deeply thankful to my parents: Rongzheng Nai and Jianlan Shen. They are the best parents that no words can express. None of this would have been possible without their unconditional love and undying support throughout my life. All that I have or hope to be, I owe to them. I am especially thankful to my wife, Lu Yin for her endless love, support, and encouragement. Thank you and I love you!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

With the emergence of data science, graph computing is becoming a crucial tool for processing big connected data. However, when mapped to modern computing systems, graph computing typically suffers from poor performance because of inefficiencies in memory subsystems. At the same time, emerging technologies, such as Hybrid Memory Cube (HMC), enable processing-in-memory (PIM) functionality, a promising technique of near-data processing (NDP), by integrating compute units in the 3D-stacked logic layer. The PIM units allows operation offloading at an instruction level, which has considerable potential to overcome the performance bottleneck of graph computing. Nevertheless, studies have not fully explored this functionality for graph workloads or identified its applications and shortcomings. The main objective of this dissertation is to enable NDP techniques for efficient graph computing. Specifically, it investigates the PIM offloading at instruction level. To achieve this goal, it presents a graph benchmark suite for understanding graph computing behaviors, and then proposes architectural techniques for PIM offloading on various host platforms.

This dissertation first presents GraphBIG, a comprehensive graph benchmark suite. To cover major graph computation types and data sources, GraphBIG selects representative data representations, workloads, and datasets from 21 real-world use cases of multiple application domains. This dissertation characterized the benchmarks on real machines and observed extremely irregular memory patterns and significant diverse behaviors across various computation types. GraphBIG helps users understand the behavior of modern graph computing on hardware architectures and enables future architecture and system research for graph computing.

To achieve better performance of graph computing, this dissertation proposes GraphPIM, a full-stack NDP solution for graph computing. This dissertation performs an analysis on modern graph workloads to assess the applicability of PIM offloading and presents hard-

ware and software mechanisms to efficiently make use of the PIM functionality. Following the real-world HMC 2.0 specification, GraphPIM provides performance benefits for graph applications without any user code modification and ISA changes. In addition, GraphPIM proposes an extension to PIM operations that can further bring performance benefits for more graph applications. The evaluation results show that GraphPIM achieves up to a 2.4× speedup with a 37% reduction in energy consumption.

To effectively utilize NDP systems with GPU-based host architectures that can fully utilize hundreds of gigabytes of bandwidth, this dissertation explores managing the thermal constraints of 3D-stacked memory cubes. Based on the real experiment with an HMC prototype, this study observes that the operating temperature of HMC is much higher than conventional DRAM, which can even cause thermal shutdown with a passive cooling solution. In addition, it also shows that even with a commodity-server cooling solution, HMC can fail to maintain the temperature of the memory dies within the normal operating range when in-memory processing is highly utilized, thereby resulting in higher energy consumption and performance overhead. To this end, this dissertation proposes CoolPIM, a thermal-aware source throttling mechanism that controls the intensity of PIM offloading on runtime. The proposed technique keeps the memory dies of HMC within the normal operating temperature using software-based techniques. The evaluation results show that CoolPIM achieves up to 1.4× and 1.37× speedups compared to non-offloading and naïve offloading scenarios.

CHAPTER 1
INTRODUCTION

1.1 The Problem: Inefficient Graph Computing on Conventional Architectures

With the emergence of data and network science, graph computing has been increasingly popular as a tool for processing large-scale network data. Nowadays, graph computing is applied to a variety of domains, such as social networks [1, 2, 3], e-commerce recommendations [4, 5], and bio-informatics [6, 7]. It is expected to become more prevalent in the future as many large-scale, real-world problems can be effectively modeled as graphs. As a result, to enable efficient graph computing, researchers have devoted a significant amount of research efforts from high-level graph analytics [1, 8] to low-level system implementations [9, 10, 11, 12, 13]. Industry has proposed multiple of graph computing platforms, such as Pregel [9] from Google, System G [13] from IBM, Trinity [14] from Microsoft, and Cassovary [15] from Twitter. Academia also has proposed a variety of graph computing systems in various hardware environments. Examples include GraphChi [10] and X-stream [16] for single machines, GraphLab [17] and GraphX [18] for distributed systems, GraphReduce [19] and Cusha [12] for GPU platforms. In addition, numerous optimization techniques have been proposed for specific graph applications [20, 21, 22, 23].

Despite all these efforts of graph platform design and graph application optimization, graph computing does not perform well on modern computing systems because of the inherent nature of the irregular connectivity between vertices, which leads to irregular memory accesses to graph data and makes the memory subsystem inefficiently utilized on both CPU- and GPU-based hardware platforms. For CPU-based platforms, it introduces high cache miss ratio and eventually results in extremely low instruction per cycle (IPC); while for GPU-based platforms, it brings memory divergence, which triggers instruction replay

and memory bandwidth underutilization, and thus reduces system throughput.

To improve the execution efficiency of graph computing, it is critical to understand the graph computing behaviors from architectural perspectives and incorporate architectural innovations for overcoming the inefficient utilization of memory subsystems. However, most of existing benchmarks target other evaluation purposes, which are less applicable for benchmarking graph computing specifically. Because of lacking sufficient benchmarking support, graph computing research requires a set of new and representative graph benchmarks and its comprehensive characterization. In addition, most existing memory subsystem techniques focus on the cache-level optimization and target only conventional applications; hence, they cannot be directly applied on graph computing systems. The graph computing problem requires solutions that utilize emerging architectural techniques and understand real-world graph computing characteristics at the same time.

1.2 The Contributions: Enabling Near-Data Processing for Efficient Graph Computing

Among the emerging technologies for memory subsystems, this dissertation investigates near-data processing (NDP) techniques, in particular, the processing-in-memory (PIM) techniques, and focuses on the question that *how to enable PIM offloading for improving graph computing efficiency*. To solve this question, this dissertation first proposes a comprehensive graph benchmark suite for understanding graph computing behavior and then proposes a novel technique that enables PIM offloading for graph computing on CPU platforms. After that, this dissertation analyzes the thermal problem of 3D-stacked memory and proposes a thermal-aware PIM offloading mechanism for graph computing on data-parallel architectures. The highlights of the contributions are as follows.

Benchmarking and characterizing graph computing: Graph computing has a wide scope with various computation types. However, most of existing benchmarks target other evaluation purposes, which are much broader than graph computing. For example, in

CloudSuite and BigDataBench, graph is only a small portion of their applications, and Graph 500 was proposed only for system performance ranking purposes. To understand graph computing properly, the graph benchmarks should cover the broad scope of graph computing and incorporate realistic frameworks, not just simplified static graph structures with no complex properties attached to vertices and edges.

To understand the behaviors of graph computing, this dissertation propose a suite of CPU/GPU graph benchmarks, GraphBIG, and analyze it on contemporary hardware. GraphBIG is inspired by the System G framework from IBM, which is a set of industrial graph computing toolkits used by many commercial clients [13]. Instead of directly using a specific industrial framework, GraphBIG abstracts one based on our experience with System G and a large number of interactions with the commercial clients. The workloads are all from representative use cases and cover all major computation types. Moreover, the framework and data representation design are both following generic techniques widely used by multiple graph systems. By ensuring the representativeness of data representations and graph workloads, GraphBIG is able to address the shortcomings of previous benchmarking efforts and achieve a generic benchmarking solution. The characterization of GraphBIG indicates high L2/L3 cache miss rates on CPUs as well as high branch/memory divergence on GPUs, and also observes diverse architectural behavior across various graph computation types and a high degree of data sensitivity.

Enabling instruction-level offloading in graph computing frameworks: The concept of NDP was initially proposed decades ago, most notably as PIM, with a variety of architecture designs and test chips [24, 25, 26, 27, 28]. However, because of fabrication issues, the widespread commercial adoption of PIM remained elusive. Recent advances in 3D stacking technology reinitiated PIM-based NDP research by integrating the logic and memory dies in the same package. Several researchers have proposed various PIM architectures and programming models [29, 30, 31], and memory vendors have also started to incorporate compute units into the memory architecture as does Hybrid Memory Cube

(HMC), proposed by Micron [32, 33]. By offloading operations on irregular data, PIM has considerable potentials of memory bandwidth saving, cache pollution reduction, and atomic instruction overhead avoidance. However, to enable PIM for graph computing, we have to address two key challenges, *what to offload* and *how to offload*. That is how to enable PIM offloading in host architectures and how to integrate the offloading feature into the software graph frameworks.

This dissertation follows the HMC 2.0 specification, which is a real-world PIM technique that will be available in the near future, and proposes a full-stack solution that enables PIM for modern graph frameworks, GraphPIM, which addresses both of the two challenges above. First, based on the key observation that the atomic access to the graph property is the main culprit for the inefficient execution of graph workloads on modern systems, GraphPIM offloads the atomic operations on the graph property to the PIM side to avoid the overhead of performing the atomic operations in the host processor as well as the inefficient utilization of the memory subsystem caused by irregular data accesses. Instead of adding new host instructions for each PIM operation, GraphPIM leverages existing host instructions to enable PIM by mapping host atomic instructions directly into PIM atomics using *uncacheable memory support* in modern architectures. Thus, with only minor architectural extensions that support PIM instruction offloading, GraphPIM significantly improves graph processing performance (up to 2.4x) without any changes in ISA and user applications.

Enabling thermal-aware offloading for graph computing on data-parallel architectures: 3D-stacking technology enables high memory bandwidth and PIM functionality; however it also raise severe thermal challenges because of its high power density and limited thermal transfer capability within the stack. Although conventional DRAMs usually stay at a low temperature with only ambient cooling, a 3D-stack can easily exceed 85C at high utilization even with an active cooling solution. Therefore, for data-parallel architectures that can consume hundreds of gigabytes of memory bandwidth, such as GPUs

and Xeon Phi, to fully utilize the 3D-stacked memory and its PIM functionality for graph computing, the PIM offloading should stay aware of the thermal limitations.

To explore managing the thermal constraints of die stacking and in-memory processing, this dissertation perform a thermal analysis on a real HMC 1.1 prototype and then model PIM functionality as in HMC 2.0 according to the real prototype results. The evaluation shows that even a commodity-server cooling could fail to maintain the memory temperature below the normal operating temperature range; so, the PIM needs to shut down for cooling down its temperature before serving memory requests again or decrease the DRAM refresh interval and frequency. Therefore, this dissertation proposes CoolPIM, a thermal-aware source throttling technique that dynamically controls the intensity of PIM offloading. Our proposed technique maintains the temperature of memory dies within the normal operating temperature with a commodity-server cooling solution, which leads to higher performance compared to naïve offloading. CoolPIM improves performance up to 1.4x and 1.37x compared to non-offloading and naïve PIM offloading without thermal consideration.

1.3 Thesis Statement

Near-data processing techniques can enable efficient graph computing by designing effective software and hardware mechanisms for architecting instruction-level processing-in-memory offloading.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 summarizes background and related work. Chapter 3 presents a comprehensive graph benchmark suite and its architectural characterization. Chapter 4 proposes a technique that enables PIM offloading in graph computing frameworks. Chapter 5 presents a thermal analysis of 3D-stacked memory and proposes a dynamic source throttling technique of thermal-aware PIM offloading for graph computing on data-parallel architectures. Chapter 6 concludes this dissertation.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 Graph Computing Platforms

As a crucial tool for processing network data, graph computing is becoming increasingly important. Several graph computing platforms have been proposed in both industrial and academic proposals.

2.1.1 Industrial Graph Platforms

A wide variety of graph platforms have been proposed by industry. In 2010, Google proposed the Pregel [9], which is a scalable graph framework for distributed environment. It follows Bulk Synchronous Parallel (BSP) and vertex-centric programming model. As an open-source counterpart of Pregel, Apach Giraph [34] was also proposed shortly after Pregel. In Pregel, the graph computation consists a series of supersteps. Each contains a computation phase and a communication phase. Active vertices perform user-defined compute function in the computation phase, while in the communication phase, messages are broadcasted through outgoing edges. GraphLab [17] is another popular graph computing framework designed for distributed environment. It is following a Gather-Apply-Scatter (GAS) model, which shares significant similarly as Pregel. In particular, GAS model pulls information from incoming edges in the Gather phase. The Apply phase then perform local computation and updates vertex states. The neighbors are further updated in the Scatter phase. GAS model can be achieved in both synchronous and asynchronous way. IBM also proposes System G, which is a comprehensive set of graph computing tools for big data [13]. System G provides a full set of toolkits, including graph visualization, graph analytics, graph middleware, and graph database. Other industrial graph frameworks include

Trinity [14] from Microsoft, Cassovary [15] from Twitter, and PGX.D [35] from Oracle.

2.1.2 Academic Graph Platforms

Multiple academic research efforts also have been proposed, focusing on various optimization targets on different hardware platforms. GraphChi [10] proposes a data management and computation scheduling method to enable large-scale graph processing on one single machine. It utilizes the Parallel Sliding Window (PSW) technique to optimize disk IO performance. With properly managed graph partitioning and sorted edgelist within partitions, GraphChi converts random accesses into a few sequential disk accesses. By utilizing the similar concept, X-stream [16] and Cusha [12] propose techniques for improving data access locality of graph computing on various platforms. Both X-stream and Cusha are following GAS programming model. Cusha follows vertex-centric GAS model, while X-stream uses edge-centric GAS model. However, Cusha is limited by the GPU memory capacity, lacking the capability of processing large-scale graphs. To solve this limitation, GraphReduce [19] proposes a hybrid method that combines vertex-centric and edge-centric GAS model. By incorporating efficient graph partitioning and data movement, GraphReduce achieves a scalable GPU framework that exceeds internal memory capacity limitation.

2.2 Graph Computing Benchmarks

Previous benchmarking efforts of graph computing are summarized in Table 2.1. Multiple existing generic benchmark suites contain graph computing workloads. As one of the most widely used benchmark suites, SPEC 2006 [44] includes two graph related workloads. Besides, multiple newly proposed benchmark suites also contain graph workloads. For example, CloudSuite [36] is a benchmark suite for cloud services. It includes one graph analytics workload, which implements the TunkRank algorithm for user ranking of twitter graphs. BigDataBench [38] is an open-source big data benchmark suite for scale-out workloads. It contains five graph workloads built on top of Hadoop/Spark framework.

Table 2.1: Comparison between GraphBIG and prior graph benchmarks. (Computation and data types are summarized in Table 3.1 and Table 3.2)

Benchmark	Graph Workloads	Framework	Data Representation	Computation Type	Data Support
SPEC int	mcf, astar	NA	Arrays	CompStruct	Data type 4
CloudSuite [36]	TunkRank	GraphLab [17]	Vertex-centric	CompStruct	Data type 1
Graph 500 [37]	Reference code	NA	CSR	CompStruct	Synthetic data
BigDataBench [38]	4 workloads	Hadoop	Tables	CompStruct	Data type 1
SSCA [39]	4 kernels	NA	CSR	CompStruct	Synthetic data
PBBS [40]	5 workloads	NA	CSR	CompStruct	Synthetic data
Parboil [41]	GPU-BFS	NA	CSR	CompStruct	Synthetic data
Rodinia [42]	3 GPU kernels	NA	CSR	CompStruct	Synthetic data
Lonestar [43]	3 GPU kernels	NA	CSR	CompStruct	Synthetic data
GraphBIG	12 CPU workloads, 8 GPU workloads	IBM System G [13]	Vertex-centric /CSR	CompStruct/ CompProp /CompDyn	All types & synthetic data

Meanwhile, BigDataBench also provides multiple social graph data and a synthetic graph generator, which is inherited from SNAP package [45]. Similarly, five graph workloads are incorporated in the Problem Based Benchmark Suite (PBBS) [40]. It is designed to be an open source repository to compare different parallel programming methodologies in terms of performance and code quality. Several GPU benchmark suites also contain graph computing workloads. For example, Parboil [41] and Rodinia [42] are targeting general GPU benchmarking. Both of them incorporate GPU graph workloads. Parboil includes BFS, while Rodinia contains BFS, PathFinder, and B+ Tree. Lonestar [43] is a GPU benchmark suites targeting irregular applications. It also includes three graph workloads, BFS, Minimum Spanning Tree, and Shortest Path.

As summarized above, most of the existing benchmark suites, including both CPU and GPU benchmark suites, target at generic evaluation purposes, which are much broader than

graph computing. Their focuses are either evaluating contemporary hardware platforms or evaluating generic computation types, such as big data and cloud computing. Because of their broad coverage, they cannot provide specialized evaluation support when comes to graph computing specifically. To overcome the limitations of generic benchmarking efforts for evaluating graph computing, graph-specific benchmarks are also proposed. As one of the most widely adopt graph benchmarking efforts, Graph 500 [37] was announced in ISC2010. The target is to establish a set of large-scale benchmarks for graph computing evaluation. Graph 500 is using BFS as the main benchmark and number of traversed edges per second (TEPS) as the key metric. Although reference code is provided, participants are encouraged to design customized software and hardware implementation to achieve highly optimized throughput. Besides, Scalable Synthetic Compact Application (SSCA) [39] is another example of graph benchmark. It provides a comprehensive graph benchmark with four kernels, including graph generation, classification, subgraph extraction, and graph clustering.

Graph computing has a broad scope, covering multiple computation types. As shown in Table 2.1, most of existing benchmarks are highly biased to graph traversal related workloads (CompStruct). The other two graph computation types, computation on dynamic graphs and on rich properties, are less studied. However, as we illustrated in the previous section, both of them are important graph computation types and cannot be overlooked when analyzing the full-scope graph computing. Moreover, without incorporating realistic frameworks, most prior graph benchmarks assume simplified static graph structures with no complex properties attached to vertices and edges. However, this is not the case for most real-world graph processing systems. The underlying framework plays a crucial role in the graph system performance. Moreover, in real-world systems, graphs are dynamic and both vertices and edges are associated with rich properties.

Multiple system-level benchmarking efforts are also ongoing for evaluating and comparing existing graph systems. Examples include LDBC benchmark, GraphBench, G.

Yong's characterization work [46], and A. L. Varbanescu's study [47]. We excluded them in the summary of Table 2.1 because of their limited usability for architectural research. In these benchmarking efforts, very few ready-to-use open-source benchmarks are provided. Detailed analysis on the architectural behaviors is also lacking.

2.3 Near-data Processing (NDP)

Because of the increasing gap between computation speed and data movement speed, also known as the "memory wall", the near-data processing (NDP) concept was intensively studied, such as intelligent controllers near memory, I/O, and disk. Among them, processing-in-memory (PIM) is the most notable one with numerous research proposals in the 1990's. Examples include computational ram [48], DIVA [25], memory-based processor array (MPA) [49], and active pages [27]. Test chips as well as systems were also proposed [24]. By placing the computation where data resides, PIM can minimize the data transfer time. However, fabrication issues limit PIM from being adopted widely in industry. On the contrary, memory vendors focus more on the innovations in memory interface techniques.

PIM is regaining research interest because of the recent advances in 3D-staking technology. With the demonstration of multiple industrial products [32, 50, 51] and academic projects [30, 31], 3D-stacking technique is already in place. In this new context of die-stacking, NDP concept, especially PIM, is revisited in recent years with a variety of research proposals.

2.3.1 3D-stacking Technology

The 3D-stacking technology enables integration of dies with different fabrication techniques. Memory and logic dies can be integrated within the same package in a cost-effective manner. Multiple industrial proposals of 3D-stacking have been presented. Examples include Micron's Hybrid Memory Cube (HMC) [32], High Bandwidth Memory

(HBM) [50] from AMD and Hynix, and IBM's Active Memory Cube (AMC) [51]. HMC consists of a single package containing multiple memory dies and one logic die [32], stacked together via TSVs. Multiple HMC packages can be connected together in a daisy-chained manner. Instead of stacking memory on top of a logic die, HBM is following an interposer-based integration, also known as 2.5D stacking. In the interposer stacking, the bandwidth limit between logic and memory is significantly smaller than 3D-stacking because of the limitation of interposer wire count. However, HBM decouples the size of logic die from memory stacks, allowing a much large memory capacity integrated into the package. AMC is a PIM architecture for exascale computing proposed by IBM. By personalizing the logic layer in HMC structure, AMC incorporates 32 lane units within the memory stack. Each lane unit is a general-purpose core with streaming vector structure.

2.3.2 PIM Architectures

Unlike the old PIM proposals in 1990s [24, 25, 26, 27, 28], which require DRAM die modifications, 3D-stacking structure provides a practical design for realizing PIM concept in a cost-effective way. It therefore enables a variety of research proposals for architecting and evaluating PIM.

Prior PIM research focuses more on fully-programmable PIMs. For example, Ahn et al. [30] proposed an in-memory programmable accelerator, named as Tesseract, for large-scale graph processing. Tesseract utilizes the 3D-stacking design of HMC and integrates in-order core to each HMC vault. With an efficient message passing mechanism and specialized hardware prefetcher, Tesseract can improve performance and energy efficiency significantly. Gao et al. proposed a PIM architecture for data analytics applications [29]. Hsieh et al. [52] also proposed a GPU-based PIM architecture that enables programmer-transparent PIM GPU systems.

In addition to fully-programmable PIM, PIM taxonomy also includes fixed-function PIMs, among which HMC is one of the examples of industrial proposals. Loh et al. [53]

compared and analyzed different taxonomies of PIM architectures. In Loh's work, PIM taxonomy is classified into three categories, non-compute, fixed-function, and programmable. Non-compute type is using logic in a transparent way. It includes build-in self test, in-stack prefetch, in-stack caching, and so on. Programmable category assumes a fully-programmable unit integrated in the logic die. The programmable unit can a general-purpose processor, such as Tesseract, or domain specific unit, such as GPU and DSP. On the other hand, fixed-function PIM provides a set of pre-defined functionality in the memory stack. The supported function can be simple operations, such as regular x86 instruction, or compound operations, such as scatter and gather operation. To our knowledge, the most relevant academic proposal of fixed-function PIM is PEI [31]. PEI incorporates a set of PIM operations in both host processor and memory. Software utilizes PIM operations by using special PIM instructions. The PIM operations will either be processed in the host processor or be offloaded to memory, according the locality monitoring result. Moreover, PEI does not bypass cache for PIM data. To ensures data coherence, PEI sends extra writebacks for the offloaded operations. Nai et al. [54] also proposed a case study of instruction-level PIM offloading for graph traversal applications. As a preliminary study, it provides application analysis without detailed performance evaluations. Various PIM architectures are also studied in the context of different application domains. For example, Zhu et al. [55] focused on data intensive applications and proposed an in-memory accelerator architecture. Similarly, Xu et al. [56] proposed a scalable PIM architecture for deep learning applications.

2.4 HMC Architecture

HMC integrates multiple DRAM dies and one logic die within a single package using die-stacking technology. As illustrated in Figure 2.1, the HMC is organized into multiple vaults that are functionally and operationally independent. The memory partitions within a vault are connected via through-silicon vias (TSVs), each of which may have multiple memory banks. Each vault incorporates a vault controller (similar to the memory controller) in the

logic layer that manages the memory partitions stacked on top of it. There is also a crossbar switch that connects all vault controllers and external I/O links. Each I/O link consists of 16 input serial lanes and 16 output serial lanes. The I/O links follow a packet-based protocol, in which the packets consist of 128-bit flow units named as FLIT [57]. Each response packet contains a tail field, which includes a 7-bit error status (ERRSTAT[6:0]). When exceeding the operational temperature limit, HMC sends back an error warning by setting the error bits to 0000001. Although HMC 2.0 only uses one thermal error, more thermal warning states can easily be defined by utilizing the unused error codes. Because of the high memory density and high-speed external links, HMC provides a dramatic improvement in memory bandwidth. In addition, HMC also opens up the possibility of supporting a variety of PIM functionality within the memory cube.

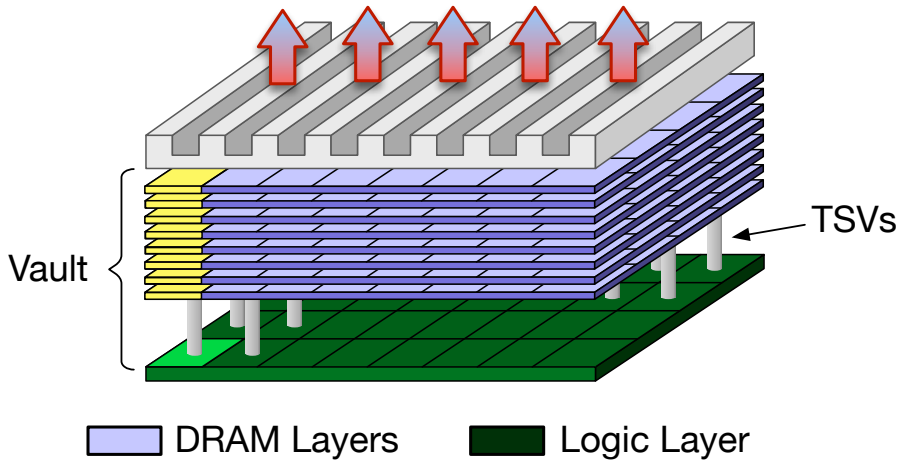


Figure 2.1: Hybrid memory cube architecture

HMC integrates multiple DRAM dies and one logic die within a single package using die-stacking technology, and thus introduces the possibility of supporting a variety of PIM functionality within the memory cube. PIM operations in HMC basically perform three steps: reading data from DRAM, performing computation on the data in the logic die, and then writing back the result to the same DRAM location. According to HMC 2.0, the PIM units perform read-modify-write (RMW) operations *atomically* within an HMC package. The corresponding DRAM bank is locked during the RMW operation, so any other memory

requests to the same bank cannot be serviced. In addition, all PIM operations include only one memory operand; the operations are performed on an immediate value and a memory operand.

Table 2.2: Atomic operations in HMC 2.0

Type	Data Size	Operation	Return
Arithmetic	8/16 byte	single/dual signed add	w/ or w/o
Bitwise	8/16 byte	swap, bit write	w/ or w/o
Boolean	16 byte	AND/NAND/OR/NOR/XOR	w/o
Comparison	8/16 byte	CAS-if equal/zero/greater/less,	w/
		compare if equal	w/

Table 2.2 lists several types of PIM operations supported by HMC 2.0: arithmetic, bitwise, boolean, and comparison. Although some operations also support eight bytes, the default data size of PIM operations is 16 bytes. Depending on the definition of specific commands, a response may or may not be returned. If the response is returned, it will include an atomic flag that indicates whether the atomic operation was successful. Depending on the commands, the original memory data may also be returned along with the response.

2.5 Thermal Challenges

Thermal analysis of 3D-stacked memory: The 3D-stacking technique enables a massive bandwidth between different dies. However, it also exposes more severe thermal challenges. Milojevic et al. [58] propose a power-efficient many-core server-on-chip system with 3D-stacked Wide I/O DRAMs [59], and conduct a thermal characterization on it. Their results show that even with embedded cores, when executing CPU-centric benchmarks, the temperature reaches 175-200 °C with less than 20W power consumption, which already exceeds the reliable memory operating temperature. However, with real cloud workloads, because of the memory-intensive feature, the power density is much below, and thus the temperatures remains at a lower range, resulting a feasible 3D-stacked system with a low-cost passive heat sink. Eckert et al. [60] explore the thermal feasibility of die-stacked PIM.

and demonstrate the thermal constraints with various processor organizations and cooling solutions. By following the recent Wide I/O DRAM technology [59], this work analyzes a PIM system similar to high-bandwidth memory (HBM) [50]. It demonstrates that even with a low-end passive heat sink, the logic die can sustain up to 8.5W while keeping the DRAM temperature below $85\,^{\circ}\mathrm{C}$. It also shows that the power budget of the logic die depends on not only the cooling solutions, but also the organization of PIM processors. Loh [61] proposes a 3D-stacked memory on top of a high-performance quad-core processor with 92W power consumption. This work demonstrates that an eight-die stacked memory plus a logic die can be kept below $95\,^{\circ}\mathrm{C}$ with an active heat sink. Similarly, Kim et al. [62] presents a programmable and scalable digital neuromorphic architecture for efficient neural computing. The proposed architecture follows a HMC-like design with 3D-stacked logic and DRAM layers. By integrating low-power processing elements (PEs) in each HMC vault, it keeps the temperature of logic die within 349K, and the temperature of DRAM dies within 344K.

Temperature impact on DRAM: Temperature affects the refresh interval and latency of DRAM cells. Guan et al. [63] analyze the temperature impact on 3D-stacked DRAMs, and propose a temperature-aware refresh mechanism for 3D-stacked DRAMs. Their mechanism adjusts the refresh rates of DRAM banks based on their actual thermal conditions at runtime. Specifically, the mechanism split the operating temperature into five ranges: 45-$65\,^{\circ}\mathrm{C}$, 65-$85\,^{\circ}\mathrm{C}$, 85-$95\,^{\circ}\mathrm{C}$, and 95-$105\,^{\circ}\mathrm{C}$. The refresh is doubled when reaching each higher temperature range. In addition, the modern DDR4 DRAMs also incorporate a similar concept for higher temperature range. In the DDR4 specification [64], the refresh rate at 85-$95\,^{\circ}\mathrm{C}$ is doubled than at a lower temperature. Liu et al. [65] also presents a dynamic DRAM refresh mechanism, which can identify and skip unnecessary refreshes using knowledge of cell retention times. By grouping DRAM rows into retention time bins and applying a different refresh rate to each bin, it refreshes only the leaky cells frequently, while refreshes most rows at a smaller frequency. Lee et al. [66] exploit the

temperature impact on DRAM latency by profiling real DRAM DIMMs. Their results demonstrate that it is possible to reduce four of the most critical timing parameters by a minimum/maximum of 17.3%/54.8% at $55\,^{\circ}\mathrm{C}$ without sacrificing correctness. This work then proposes an adaptive-latency DRAM, a mechanism that adaptively reduces the timing parameters for DRAM modules based on the current operating condition. The proposed mechanism can improve the performance of memory-intensive workloads by an average of 14% without introducing any errors.

CHAPTER 3

GRAPHBIG: UNDERSTANDING GRAPH COMPUTING

3.1 Introduction

Graph computing is comprised of multiple research areas, from low level architecture design to high level data mining and visualization algorithms. Enormous research efforts across multiple communities have been invested in this discipline [1]. However, researchers focus more on analyzing and mining graph data, while paying relatively less attention to the performance of graph computing [21, 3]. Although high performance implementations of specific graph applications and systems exist in prior literature, a comprehensive study on the full spectrum of graph computing is still missing [20, 22]. Unlike prior work focusing on graph traversals and assuming simplified data structures, graph computing today has a much broader scope. In today's graph applications, not only has the structure of graphs analyzed grown in size, but the data associated with vertices and edges has become richer and more dynamic, enabling new hybrid content and graph analysis [8]. Besides, the computing platforms are becoming heterogeneous. More than just parallel graph computing on CPUs, there is a growing field of graph computing on Graphic Processing Units (GPUs).

The challenges in graph computing come from multiple key issues like frameworks, data representations, computation types, and data sources [8]. First, most of the industrial solutions deployed by clients today are in the form of an integrated framework [67, 68, 13]. In this context, elementary graph operations, such as find-vertex and add-edge are part of a rich interface supported by graph datastructures and they account for a large portion of the total execution time, significantly impacting the performance. Second, the interaction of data representations with memory subsystems greatly impacts performance. Third, although graph traversals are considered to be representative graph applications, in practice,

graph computing has a much broader scope. Typical graph applications can be grouped into three computation types: (1) computation on graph structure, (2) computation on rich properties, and (3) computation on dynamic graphs. Finally, as a data-centric computing tool, graph computing is sensitive to the structure of input data. Several graph processing frameworks have been proposed lately by both academia and industry [67, 17, 34, 13, 16, 12]. Despite the variety of these frameworks, benchmarking efforts have focused mainly on simplified static memory representations and graph traversals, leaving a large area of graph computing unexplored [37, 69]. Little is known, for example, about the behavior of full-spectrum graph computing with dynamic data representations. Likewise, graph traversal is only one computation type. What is the behavior of other algorithms that build graphs or modify complex properties on vertices and edges? How is the behavior of graph computing influenced by the structure of the input data? To answer these questions, we have to analyze graph workloads across a broader spectrum of computation types and build our benchmarks with extended data representations.

To understand the full-spectrum of graph computing, we propose a benchmark suite, *GraphBIG*[1], and analyze it on contemporary hardware. GraphBIG is inspired by IBM's System G framework, which is a comprehensive set of industrial graph computing toolkits used by many commercial clients [13]. Based on our experience with a large set of real world user problems, we selected representative graph data representations, interfaces, and graph workloads to design our GraphBIG benchmark suite. GraphBIG utilizes a dynamic, vertex-centric data representation, which is widely utilized in real-world graph systems, and selects workloads and datasets inspired by use cases from a comprehensive selection of application domains. By ensuring the representativeness of data representations and graph workloads, GraphBIG is able to address the shortcomings of previous benchmarking efforts and achieve a generic benchmarking solution. The characterization of performance on GraphBIG workloads can help researchers understand not only the architectural behaviors

[1]GraphBIG is open-sourced under BSD license. The source codes, datasets, and documents are released in our github repository (http://github.com/graphbig/graphBIG).

of specific graph workloads, but also the trend and correlations of full-spectrum graph computing.

In this chapter, we first discuss and summarize the key factors of graph computing, and then illustrates the methodology and workloads of our proposed GraphBIG. In Section 3.4, we characterize the workloads from multiple perspectives on CPU/GPU hardware. Finally, in Section 3.5, we summarize our work.

3.2 Graph Computing: Key Factors

Although graph traversals, such as Breadth-first Search and Shortest-path, are usually considered as representative graph applications, real-world graph computing also performs various other comprehensive computations. In real-world practices, graph computing contains a broad scope of use cases, from cognitive analytics to data exploration. The wide range of use cases introduces not only unique, but also diverse features of graph computing. The uniqueness and diversity are reflected in multiple key factors, including frameworks, data representations, computation types, and data sources. To understand graph computing in a holistic way, we first analyze these key factors of graph computing in this section.

Framework: Unlike standalone prototypes of graph algorithms, graph computing systems largely rely on specific frameworks to achieve various functionalities. By hiding the details of managing both graph data and requests, the graph frameworks provide users primitives for elementary graph operations. The examples of graph computing frameworks include GraphLab [17], Pregel [9], Apache Giraph [34], and IBM System G [13]. They all share significant similarity in their graph models and user primitives. First, unlike simplified algorithm prototypes, graph systems represent graph data as a *property graph*, which associates user-defined properties with each vertex and edge. The properties can include meta-data (e.g., user profiles), program states (e.g., vertex status in BFS or graph coloring), and even complex probability tables (e.g., Bayesian inference). Second, instead of directly operating on graph data, the user defined applications achieve their algorithms via

framework-defined primitives, which usually include find/delete/add vertices/edges, traverse neighbours, and update properties.

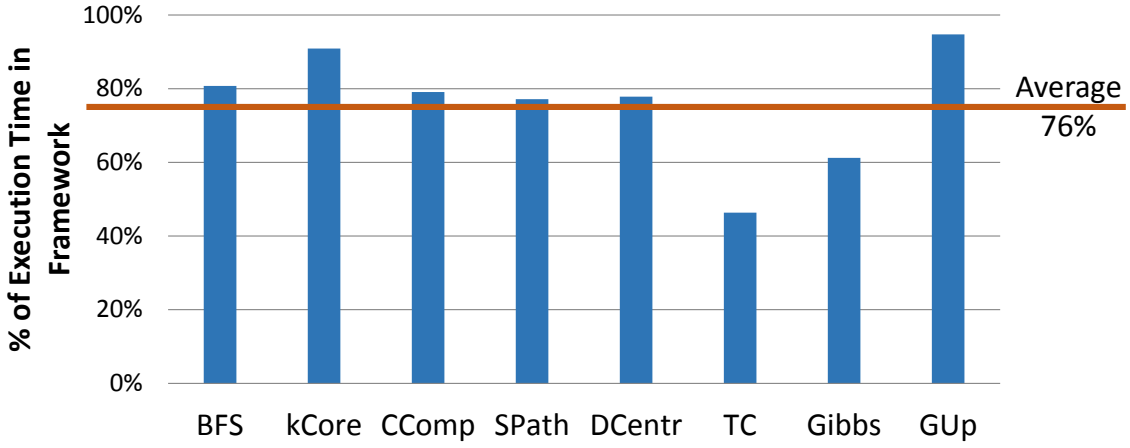


Figure 3.1: Execution time of framework

To estimate the framework's impact on the graph system performance, we performed profiling experiments on a series of typical graph workloads with IBM System G framework. As shown in Figure 3.1, a significant portion of time is contributed by the framework for most workloads, especially for graph traversal based ones. On average, the in-framework time is as high as 76%. It clearly shows that the heavy reliance on the framework indeed results in a large portion of in-framework execution time. It can bring significant impacts on the architecture behaviors of the upper layer graph workloads. Therefore, to understand graph computing, it is not enough to study only simple standalone prototypes. Workload analysis should be performed with representative frameworks, in which multiple other factors, such as flexibility and complexity, are considered, leading to design choices different from academic prototypes.

Data representation: Within the graph frameworks, various data representations can be incorporated for organizing in-memory graph data. The differences between in-memory data representations can significantly affect the architectural behaviors, especially memory subsystem related features, and eventually impact the overall performance.

One of the most popular data representation structure is Compressed Sparse Row (CSR).

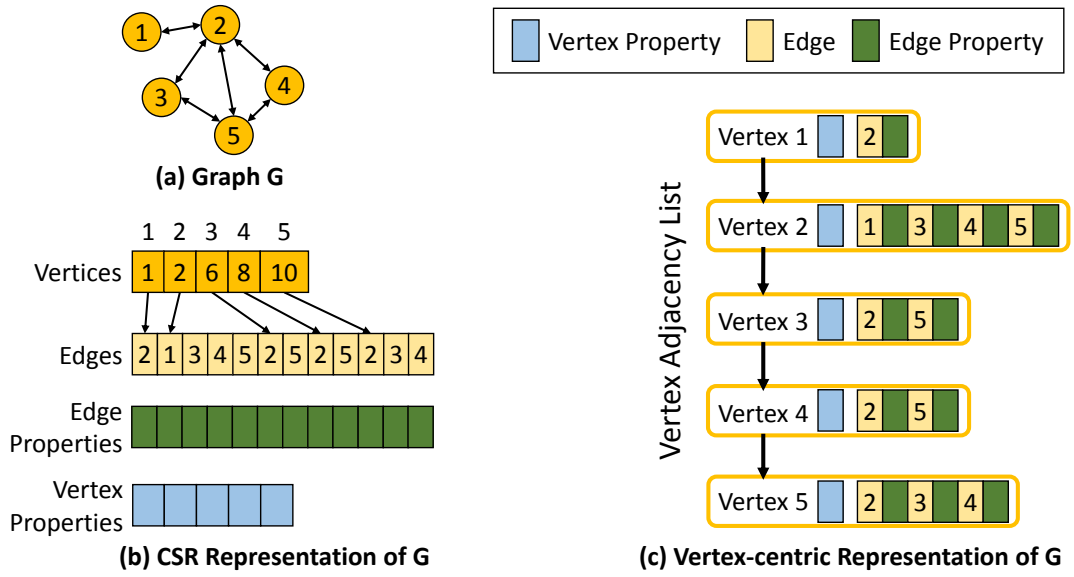


Figure 3.2: Illustration of data representations. (a) graph G, (b) its CSR representation, and (c) its vertex-centric representation.

As illustrated in Figure 3.2(a)(b), CSR organizes vertices, edges, and properties of graph G in separate compact arrays. (Variants of CSR also exist. For example, Coordinate List (COO) format replaces the vertex array in CSR with an array of source vertices of each edge.) The compact format of CSR saves memory size and simplifies graph build/copy/transfer complexity. Because of its simplicity, CSR is widely used in the literature. However, its drawback is also obvious. CSR is only suitable for static data with no structural updates. This is the case for most graph algorithm prototypes. Nevertheless, real-world graph systems usually are highly dynamic in both topologies and properties. Thus, more flexible data representations are incorporated in graph systems. For example, IBM System G, as well as multiple other frameworks, is using a vertex-centric structure, in which a vertex is the basic unit of a graph. As shown in Figure 3.2(c), the vertex property and the outgoing edges stay within the same vertex structure. Meanwhile, all vertices form up an adjacency list with indices. Although the compact format of CSR may bring better locality and lead to better cache performance, graph computing systems usually utilize vertex-centric structures because of the flexibility requirement of real-world use cases [17, 13].

Table 3.1: Graph computation type summary

Graph Computation Type	Feature	Example
Computation on graph structure (CompStruct)	Irregular access pattern, heavy read accesses	BFS traversal
Computation on graphs with rich properties (CompProp)	Heavy numeric operations on properties	Belief propagation
Computation on dynamic graphs (CompDyn)	Dynamic graph, dynamic memory footprint	Streaming graph

Table 3.2: Graph data source summary

No.	Graph Data Source	Example	Feature
1	Social(/economic/political) network	Twitter graph	Large connected components, small shortest path lengths
2	Information(/knowledge) network	Knowledge graph	Large vertex degrees, large small hop neighbourhoods
3	Nature(/bio/cognitive) network	Gene network	Complex properties, structured topology
4	Man-made technology network	Road network	Regular topology, small vertex degrees

Computation types: Numerous graph applications exist in previous literature and real-world practices. Despite the variance of implementation details, generally, graph computing applications can be classified into a few computation types [70]. As shown in Table 3.1, we summarize the applications into three categories according to their different computation targets: graph structure, graph properties, and dynamic graphs. They have different features in terms of read/write/numeric intensity. (1) Computation on the graph structure incorporates a large number of memory accesses and limited numeric operations. Their irregular memory access pattern leads to extremely poor spatial locality. (2) On the contrary, computation on graphs with rich properties introduces lots of numeric computations on properties, which leads to hybrid workload behaviors. (3) For computation on dynamic graphs, it also shows an irregular pattern as the first computation type. However, the updates of graph structure lead to high write intensity and dynamic memory footprint.

Graph data sources: As a data-centric computing tool, graph computing heavily relies

on data inputs. As shown in Table 3.2, we summarize graph data into four sources [70]. The social network represents the interactions between individuals/organizations. The key features of social networks include high degree variances, small shortest path lengths, and large connected components [2]. On the contrary, an information network is a structure, in which the dominant interaction is the dissemination of information along edges. It usually shows large vertex degrees, and large two-hop neighbourhoods. The nature network is a graph of biological/cognitive objects. Examples include gene network [71], deep belief network (DBN) [72] and biological network [73].They typically incorporate structured topologies and rich properties addressing different targets. Man-made technology networks are formed by specific man-made technologies. A typical example is a road network, which usually maintains small vertex degrees and a regular topology.

3.3 GraphBIG: Benchmarking Graph Computing

3.3.1 Methodology

To understand the graph computing, we propose GraphBIG, a benchmark suite inspired by IBM System G, which is a comprehensive set of graph computing tools, cloud, and solutions for Big Data [13]. GraphBIG includes representative benchmarks from both CPU and GPU sides to achieve a holistic view of general graph computing.

Framework: To represent real-world scenarios, GraphBIG utilizes the framework design and data representation inspired by IBM System G, which is a comprehensive graph computing toolsets used by several real-world scenarios. Like many other industrial solutions, the major concerns of SystemG design include not only performance, but also flexibility, complexity, and usability. System G framework enables us to utilize its rich use case and dataset support and summarize workloads from one of the representative industrial graph solutions. By ensuring the representativeness of workloads and data representations, GraphBIG help users understand the full-spectrum graph computing. Like several other graph systems, GraphBIG follows the vertex-centric data representation, in which a ver-

tex is the basic unit of a graph. The vertex property and the outgoing edges stay within the same vertex structure. All vertices' structures form an adjacency list and the outgoing edges inside the vertex structure also form an adjacency list of edges. The graph computing workloads are implemented via framework primitives, such as find/add/delete vertex/edge and property update.By linking the same core code with various CUDA kernels, the GPU benchmarks are also utilizing the same framework. In addition, because of the nature of GPU computing, the graph data in GPU memory is organized as CSR/COO structure. In the graph populating step, the dynamic vertex-centric graph data in CPU main memory is converted and transferred to GPU side. Moreover, by replacing System G's commercial components with rewritten source codes, we are able to open source GraphBIG for public usage under BSD license.

Workload Selection: GraphBIG follows the workflow shown in Figure 3.3. By analyzing real-world use cases from IBM System G customers, we summarize computation types and graph data types. Meanwhile, we select workloads and datasets according to their popularity (use frequency). To ensure the coverage, we then reselect the workloads and datasets to cover all computation and data types. After that, we finalize the workloads and datasets to form our GraphBIG benchmark suite. In this way, the representativeness and coverage are addressed at the same time.

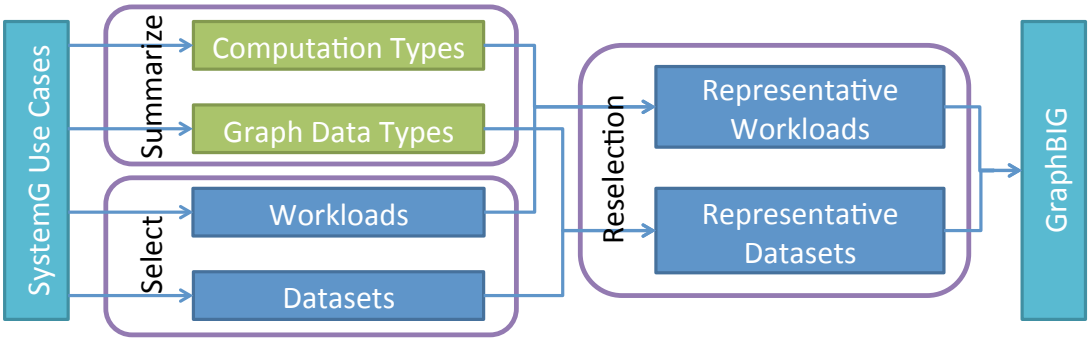


Figure 3.3: GraphBIG workload selection flow

To understand real-world graph computing, we analyzed 21 key use cases of graph computing from multiple application domains. The use cases are all from real-world practices

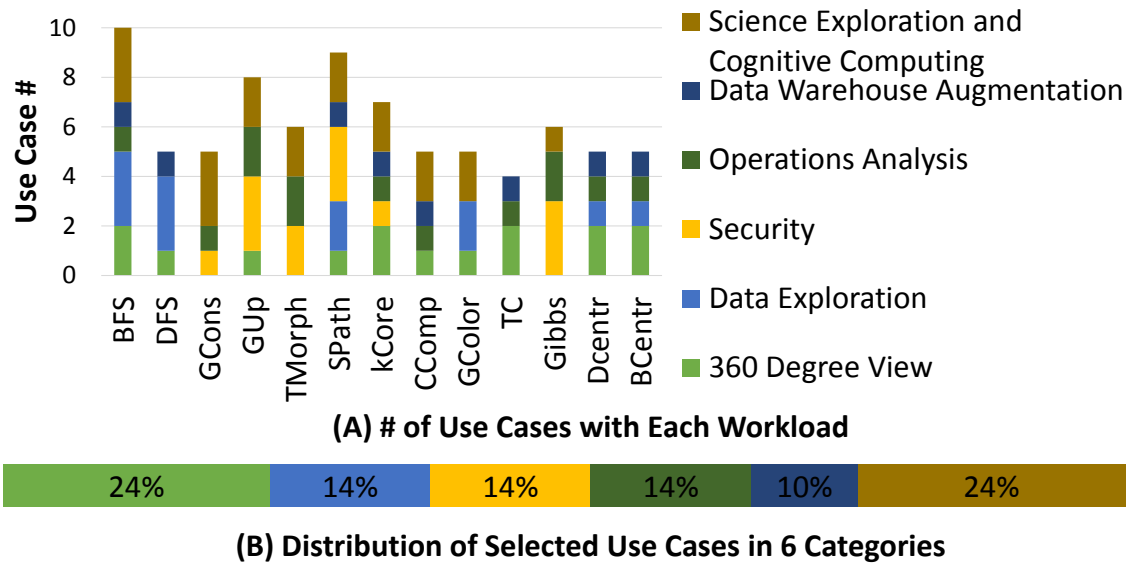


Figure 3.4: Real-world use case analysis

of IBM System G customers [13, 70]. As shown in Figure 3.4, the use cases come from six different categories, from cognitive computing to data exploration. Their percentage in each category is shown in Figure 3.4(B), varying from 24% to 10%. Each use case involves multiple graph computing algorithms. As explained previously, we then select representative workloads from the use cases according to the number of used times. Figure 3.4(A) shows the number of use cases of each chosen workload with the breakdown by categories. The most popular workload, BFS, is used by 10 different use cases, while the least popular one, TC, is also used by 4 use cases. From Figure 3.4(A), we can see that the chosen workloads are all widely used in multiple real-world use cases. After the summarize step, a reselection is performed in merge step to cover all computation types.

3.3.2 Benchmark Description

As explained in Figure 3.3 and Figure 3.4, we analyze real-world use cases and then select workloads by considering the key factors together. The workloads in our proposed GraphBIG are summarized in Table 3.3. For explanation purpose, we group the workloads into four categories according to their high level usages. The details are further explained

Table 3.3: GraphBIG workload summary

Category	Workload	Computation Type	CPU	GPU	Use Case Example
Graph traversal	BFS	CompStruct	√	√	Recommendation for Commerce
	DFS	CompStruct	√		Visualization for Exploration
Graph update	Graph construction (GCons)	CompDyn	√		Graph Analysis for Image Processing
	Graph update (GUp)	CompDyn	√		Fraud Detection for Bank
	Topology morphing (TMorph)	CompDyn	√		Anomaly Detection at Multiple Scales
Graph analytics	Shortest path (SPath)	CompStruct	√	√	Smart Navigation
	K-core decomposition (kCore)	CompStruct	√	√	Large Cloud Monitoring
	Connected component (CComp)	CompStruct	√	√	Social Media Monitoring
	Graph coloring (GColor)	CompStruct		√	Graph matching for genomic medicine
	Triangle count (TC)	CompProp	√	√	Data Curation for Enterprise
	Gibbs inference (GI)	CompProp	√		Detecting Cyber Attacks
Social analysis	Degree centrality (DCentr)	CompStruct	√	√	Social Media Monitoring
	Betweenness centrality (BCentr)	CompStruct	√	√	Social Network Analysis in Enterprise

below.

Graph traversal: Graph traversal is the most fundamental operation of graph computing. Two workloads – Breadth-first Search (BFS) and Depth-first Search (DFS) are selected. Both are widely-used graph traversal operations.

Graph construction/update: Graph update workloads are performing computations on dynamic graphs. Three workloads are included as following. (1) Graph construction (GCons) constructs a directed graph with a given number of vertices and edges. (2) Graph update (GUp) deletes a given list of vertices and related edges from a existing graph. (3)

Topology morphing (TMorph) generates an undirected moral graph from a directed-acyclic graph (DAG). It involves graph construction, graph traversal, and graph update operations.

Graph analytics: There are three groups of graph analytics, including topological analysis, graph search/match, and graph path/flow. Since basic graph traversal workloads already cover graph search behaviors, here we focus on topological analysis and graph path/flow. As shown in Table 3.3, five chosen workloads cover the two major graph analytic types and two computation types. The shortest path is a tool for graph path/flow analytics, while the others are all topological analysis. In their implementations, the shortest path is following Dijkstra's algorithm. The k-core decomposition is using Matula & Beck's algorithm [74]. The connected component is implemented with BFS traversals on the CPU side and with Soman's algorithm [75] on the GPU side. The triangle count is based on Schank's algorithm [76] and the graph coloring is following Luby-Jones' proposal [77]. Besides, the Gibbs inference is performing Gibbs sampling for approximate inference in bayesian networks.

Social analysis: Due to its importance, social analysis is listed as a separate category in our work, although generally social analysis can be considered as a special case of generic graph analytics. We select graph centrality to represent social analysis workloads. Since closeness centrality shares significant similarity with shortest path, we include the betweenness centrality with Brandes' algorithm [78] and degree centrality [79].

3.3.3 Graph data support

To address both representativeness and coverage of graph data sets, we consider two types of graph data, real-world data and synthetic data. Both are equally important, as explained in Section 3.2. The real-world data sets can illustrate real graph data features, while the synthetic data can help in analyzing workloads because of its flexible data size and relatively short execution time. Meanwhile, since the focus of our work is the architectural impact of graph computing, the dataset selection should not bring obstacles for architectural charac-

terizations on various hardware platforms. Large datasets are infeasible for small-memory platforms because of their huge memory footprint sizes. Therefore, we only include one large graph data in our dataset selection.As shown in Table 3.4, we collect four real-world data sets and a synthetic data set to cover the requirements of both sides. The details of the chosen data sets are explained below. All data sets are publicly available in our github wiki.

Table 3.4: Graph data set summary

Data Set	**Type**	**Vertex#**	**Edge#**
Twitter Graph	Type 1	120M	1.9B
IBM Knowledge Repo	Type 2	154K	1.72M
IBM Watson Gene Graph	Type 3	2M	12.2M
CA Road Network	Type 4	1.9M	2.8M
LDBC Graph	Synthetic	Any	Any

(1) Real-world data: Four real-world graph data sets are provided, including twitter graph, IBM knowledge Repo, IBM Watson Gene Graph, and CA road network. The vertex/edge numbers of each data set are shown in Table 3.4. The twitter graph is a pre-processed data set of twitter transactions. In this graph, twitter users are the vertices and twit/retwit communications form the edges. In IBM Knowledge Repo, two types of vertices, users and documents, form up a bipartite graph. An edge represents a particular document is accessed by a user. It is from a document recommendation system used by IBM internally. As an example of bio networks, the IBM Watson Gene graph is a data set used for bioinformatic research. It is representing the relationships between gene, chemical, and drug. The CA road network is a network of roads in California [45]. Intersections and endpoints are represented by nodes and the roads connecting these intersections or road endpoints are represented by undirected edges.

(2) Synthetic data: The LDBC graph is a synthetic data set generated by LDBC data generator and represents social network features [80]. The generated LDBC data set can have arbitrary data set sizes while keeping the same features as a facebook-like social net-

work. The LDBC graph enables the possibility to perform detailed characterizations of graph workloads and compare the impact of data set size.

3.4 GraphBIG Characterization and Evaluation

3.4.1 Characterization Methodology

Hardware configurations: We perform our experiments on an Intel Xeon machine with Nvidia Tesla K40 GPU. The hardware and OS details are shown in Table 3.5. To avoid the uncertainty introduced by OS thread scheduling, we schedule and pin threads to different hardware cores.

Table 3.5: Test machine configurations

Processor	Type	Xeon E5-2670
	Frequency	2.6 GHz
	Core #	2 sockets x 8 cores x 2 threads
	Cache	32 KB L1, 256 KB L2, 20 MB L3
	MemoryBW	51.2 GB/s (DDR3)
GPU	Type	Nvidia Tesla K40
	CUDA Core	2880
	Memory	12 GB
	MemoryBW	288 GB/s (GDDR5)
	Frequency	Core-745 MHz Memory-3 GHz
Host System	Memory	192 GB
	Disk	2 TB HDD
	OS	Red Hat Enterprise Linux 6

Table 3.6: Graph data in the experiments

Experiment Data Set	**Vertex #**	**Edge #**
Twitter Graph (sampled)	11M	85M
IBM Knowledge Repo	154K	1.72M
IBM Watson Gene Graph	2M	12.2M
CA Road Network	1.9M	2.8M
LDBC Graph	1M	28.82M

Datasets: In the characterization experiments, we first use synthetic graph data to enable in-depth analysis for multiple architectural features of both CPU and GPU sides. As

shown in Table 3.6, the LDBC graph with 1 million vertices is selected. Four real-world data sets are then included for data sensitivity studies. The Twitter graph is sampled in our experiments because of the extremely large size of the original graph. In our experiments, although the test platform incorporates a large memory capacity on CPU side, the GPU side has only 12GB memory, which limits the dataset size. Thus, huge size datasets are infeasible in the experiments. Moreover, we intentionally select datasets from diverse sources to cover different graph types. With the combination of different graph sizes and types, the evaluation can illustrate the data impact comprehensively. In addition, because of the special computation requirement of Gibbs Inference workload, the bayesian network MUNIN [81] is used. It includes 1041 vertices, 1397 edges, and 80592 parameters.

Profiling method: In our experiments, the hardware performance counters are used for measuring detailed hardware statistics. In total, around 30 hardware counters of the CPU side and 25 hardware metrics of the GPU side are collected. For the profiling of CPU benchmarks, we designed our own profiling tool embedded within the benchmarks. It is utilizing the perf_event interface of Linux kernel for accessing hardware counters and the libpfm library for encoding hardware counters from event names. For GPU benchmarks, the nvprof tool from Nvidia CUDA SDK is used.

Metrics for CPUs: In the experiments, we are following a hierarchical profiling strategy. Multiple metrics are utilized to analyze the architectural behaviors.

For the CPU benchmarks, *Execution cycle breakdown* is first analyzed to figure out the bottleneck of workloads. The breakdown categories include frontend stall, backend stall, retiring, and bad speculation. In modern processors, frontend includes instruction fetching, decoding, and allocation. After allocated, backend is responsible for instruction renaming, scheduling, execution, and commit. It also involves memory subsystems. *Cache MPKI* is then analyzed to understand memory subsystem behaviors. We estimated the MPKI values of L1D, L2, and LLC. In addition, we also measured multiple other metrics, including *IPC, branch miss rate, ICache miss rate, and DTLB penalty*. These metrics cover major

architectural factors of modern processors.

Metrics for GPUs: For the GPU side experiments, we first analyzed the divergence of given benchmarks. Two metrics are measured, one is *branch divergence rate* (BDR) and another is *memory divergence rate* (MDR). We use the following equations to express the degree of branch and memory divergence.

$$\text{branch divergence rate (BDR)} = \frac{\text{inactive threads per warp}}{\text{warp size}}$$

$$\text{memory divergence rate (MDR)} = \frac{\text{replayed instructions}}{\text{issued instructions}}$$

BDR is the average ratio of inactive threads per warp, which is typical caused by divergent branches. MDR is the fraction of issued instructions that are replayed. In modern GPUs, a load or store instruction would be replayed if there is a bank conflict or the warp accesses more than one 128-byte block in memory. The replay may happen multiple times until all the requested data have been read or written. Thus, we estimate the memory divergence by measuring the number of replayed instructions. Both BDR and MDR range from 0 to 1 with higher value representing higher divergence.

3.4.2 CPU Characterization Results

Workload Characterization

In this section, we characterize GraphBIG CPU workloads with a top-down characterization strategy. The results are explained as following.

Execution time breakdown: The execution time breakdown is shown in Figure 3.5 and grouped by computation types. The Frontend and Backend represent the frontend bound and backend bound stall cycles respectively. The BadSpeculation is the cycles spent on wrong speculations, while the Retiring is the cycles of successfully retired instructions. It is a common intuition that irregular data accesses are the major source of inefficiencies of graph computing. The breakdown of execution time also supports such intuition. It

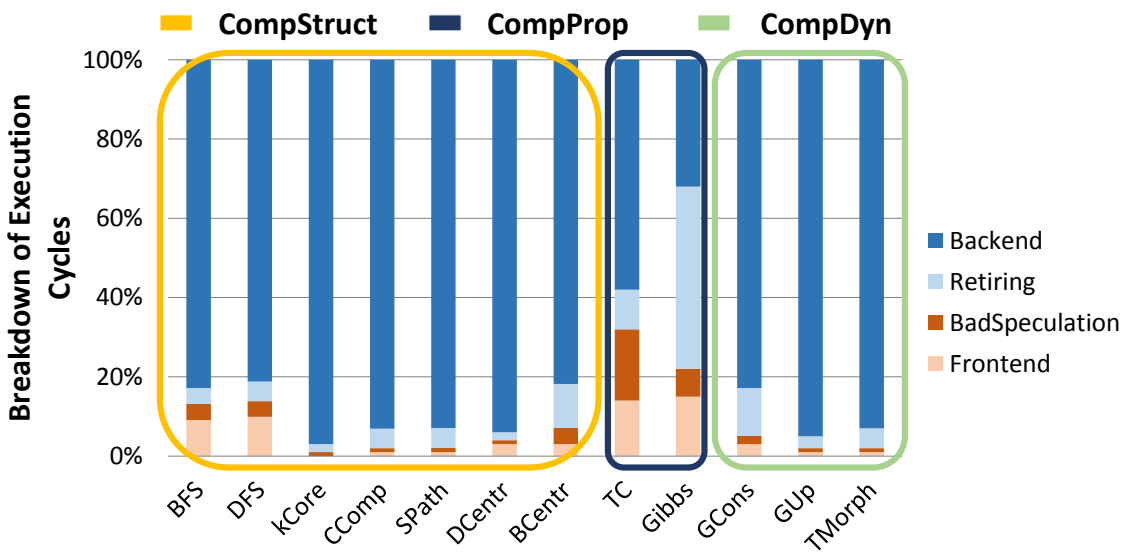


Figure 3.5: Execution time breakdown of GraphBIG CPU workloads

is shown that the backend indeed takes dominant time for most workloads. In extreme cases, such as kCore and GUp, the backend stall percentage can be even higher than 90%. However, different from the simple intuition, the outliers also exist. For example, the workloads of computation on rich properties (CompProp) category shows only around 50% cycles on backend stalls. The variances between computation types further demonstrates the necessity of covering different computation types.

Core analysis: Although execution stall can be triggered by multiple components in the core architecture, instruction fetch and branch prediction are usually the key inefficiency sources. Generally, a large number of ICache misses or branch miss predictions can significantly affect architectural performance, because modern processors usually don't incorporate efficient techniques to hide ICache/branch related penalties. In previous literatures, it was reported that many big data workloads, including graph applications, suffer from high ICache miss rate [36]. However, in our experiments, we observe different outcomes.As shown in Figure 3.6, the ICache MPKI of each workload all show below 0.7 values, though small variances still exist. The different ICache performance values are resulted from the design differences of the underlying frameworks. Open-source big data frameworks typi-

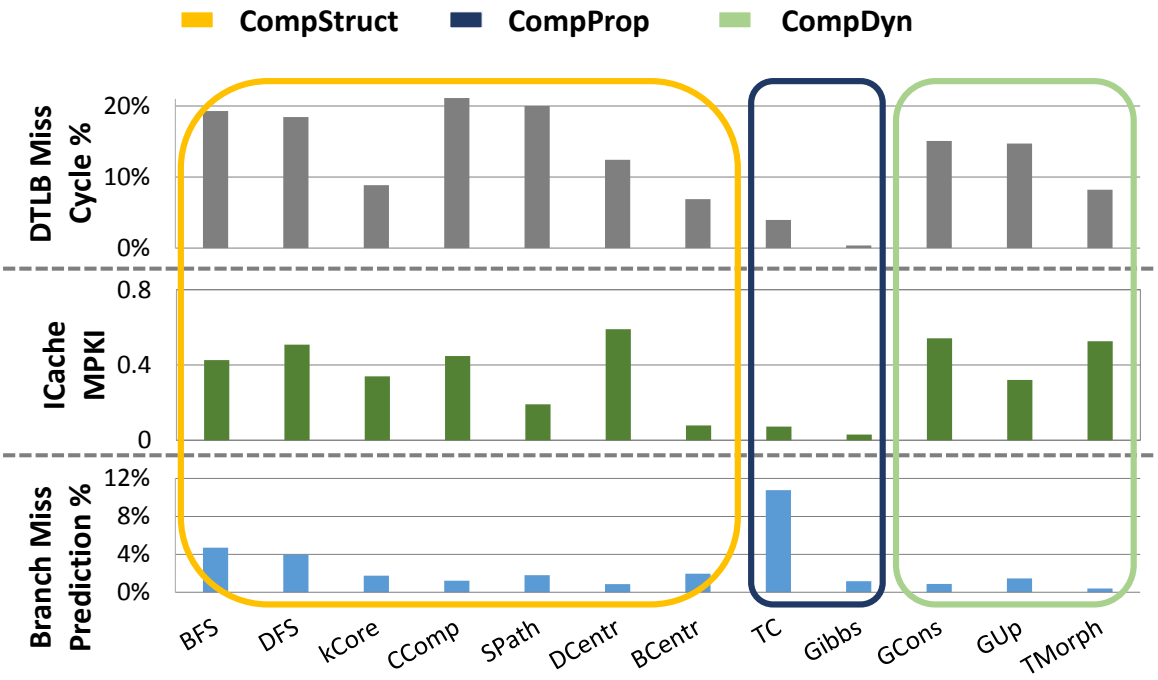


Figure 3.6: DTLB penalty, ICache MPKI, and branch miss rate of GraphBIG CPU workloads

cally incorporate many other existing libraries and tools. Meanwhile, the included libraries may further utilize other libraries. Thus, it eventually results in deep software stacks, which lead to complex code structures and high ICache MPKI. However, in GraphBIG, very few external libraries are included and a flat software hierarchy is incorporated. Because of its low code structure complexity, GraphBIG shows a much lower ICache MPKI.

The branch prediction also shows low miss prediction rate in most workloads except for TC, which reaches as high as 10.7%. The workloads from other computation types show a miss prediction rate below 5%. The difference comes from the special intersection operations in TC workload. It is also in accordance with the above breakdown result, in which TC consumes a significant amount of cycles in BadSpeculation.

The DTLB miss penalty is shown in Figure 3.6. The cycles wasted on DTLB misses is more than 15% of totaly execution cycles for most workloads. On average, it still takes 12.4%. The high penalty is caused by two sources. One is the large memory footprint of graph computing applications, which cover a large number of memory pages. Another

is the irregular access pattern, which incorporates extremely low page locality. Diversity among workloads also exists. The DTLB miss penalty reaches as high as 21.1% for Connected Component and as low as 3.9% for TC and 1% for Gibbs. This is because for computation on properties, the memory accesses are centralized within the vertices. Thus, low DTLB-miss penalty time is observed.

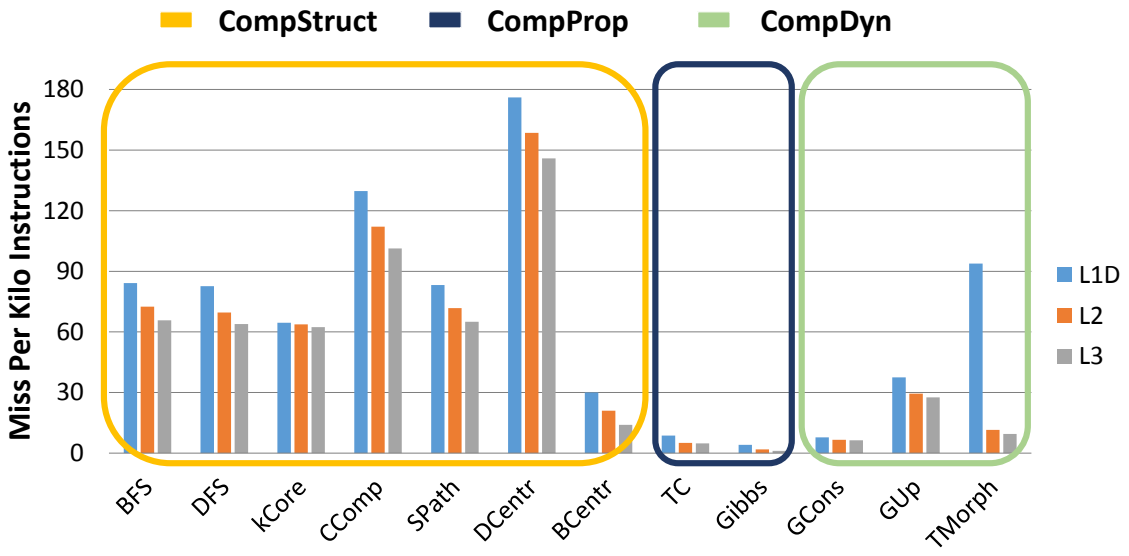


Figure 3.7: Cache MPKI of GraphBIG CPU workloads

Cache performance: As shown in previous sections, cache plays a crucial role in graph computing performance. In Figure 3.7, the MPKI of different levels of caches are shown. On average, a high L3 MPKI is shown, reaching as high as 48.77. Degree Centrality and Connected Component show even higher MPKI, which are 145.9 and 101.3 respectively. For computations on the graph structure (CompStruct), a generally high MPKI is observed. On the contrary, CompProp shows an extremely small MPKI value compared with other workloads. This is in accordance with its computation features, in which memory accesses happen mostly inside properties with a regular pattern. The workloads of computation on dynamic graphs (CompDyn) introduce diverse results, ranging from 6.3 to 27.5 in L3 MPKI. This is because of the diverse operations of each workload. GCons adds new vertices/edges and sets their properties one by one, while GUp mostly deletes them in a random

manner. In GCons, significantly better locality is observed because each new vertex/edge will be immediately reused after insertion. The TMorph involves graph traversal, insertion, and deletion operations. Meanwhile, unlike other workloads, TMorph includes no small size local queues/stacks, leading to a high MPKI in L1D cache. However, its graph traversal pattern results in relatively good locality in L2 and L3.

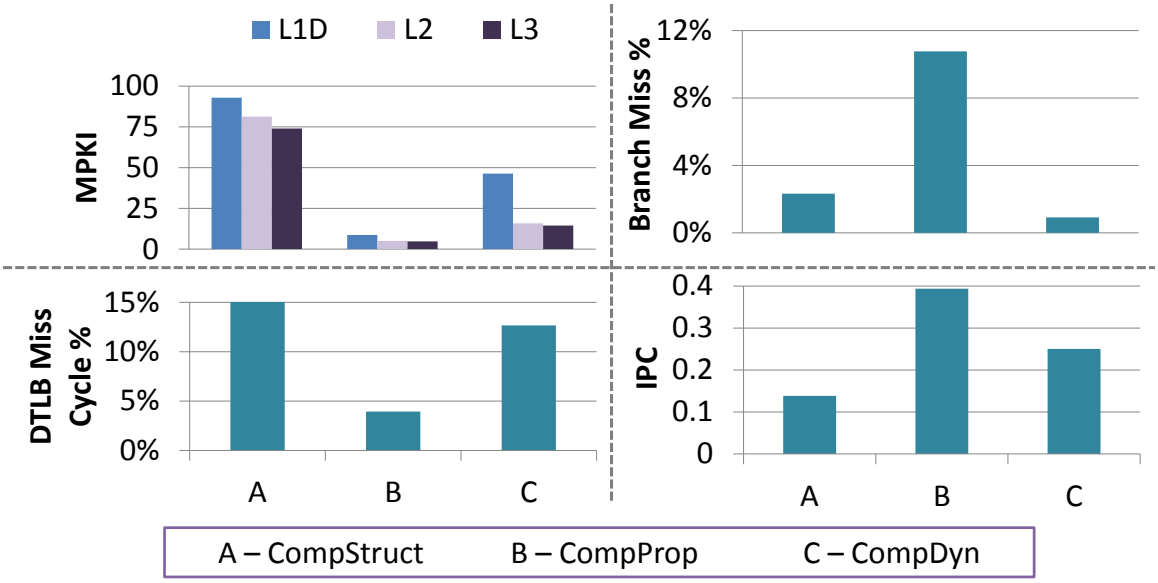


Figure 3.8: Average behaviors of GraphBIG CPU workloads by computation types

Computation type behaviors: The average behaviors of each computation type are shown in Figure 3.8. Although variances exist within each computation type, the average results demonstrate their diverse features. The CompStruct shows significantly higher MPKI and DTLB miss penalty values because of its irregular access pattern when traversing through graph structure. Low and medium MPKI and DTLB values are shown in CompProp and CompDyn respectively. Similarly, the CompProp suffers from a high branch miss rate while other two types do not. In the IPC results, CompStruct achieves the lowest IPC value due to the penalty from cache misses. On the contrary, CompProp shows the highest IPC value. The IPC value of CompDyn stays between them. Such feature is in accordance with their access patterns and computation types.

Data sensitivity: To study the impact of input data sets, we performed experiments on

four real-world data sets from different types of sources and the LDBC synthetic data (We excluded the workloads that cannot take all input datasets).

Figure 3.9: Cache hit rate, DTLB penalty, and IPC of GraphBIG CPU workloads with different data sets

Despite the extremely low L2/L3 hit rates, Figure 3.9 shows relatively higher L1D hit rates for almost all workloads and data sets. This is because graph computing applications all incorporate multiple small size structures, such as task queues and temporal local variables. The frequently accessed meta data introduces a large amount of L1D cache hits except for DCentr, in which there is a only limited amount of meta data accesses. From the results in Figure 3.9, we can also see that twitter data shows highest DTLB miss penalty in most workloads. Such behavior eventually turns into lowest IPC values in most workloads except SPath, in which higher L1D cache hit rate of the twitter graph helps performance

significantly. Triangle Count (TC) achieves highest IPC with the knowledge data set, because of its high L2/L3 hit rate and low TLB penalty. The high L3 hit rate of the watson data also results in a high IPC value. However, the twitter graph's high L3 hit rate is offsetted by its extremely high DTLB miss cycles, leading to the lowest IPC value. The diversity is caused by the different topological and property features of the real-world data sets. It is clearly shown that significant impacts are introduced by the graph data on overall performance and other architectural features.

Observations

In the characterization experiments, by measuring several architectural factors, we observed multiple graph computing features. The key observations are summarized as following.

- Backend is the major bottleneck for most graph computing workloads, especially for CompStruct category. However, such behavior is much less significant in CompProp category.

- The ICache miss rate of GraphBIG is as low as conventional applications, unlike many other big data applications, which are known to have high ICache miss rate. This is because of the flat code hierarchy of the underlying framework.

- Graph computing is usually considered to be cache-unfriendly. L2 and L3 caches indeed show extremely low hit rates in GraphBIG. However, L1D cache shows significantly higher hit rates. This is because of the locality of non-graph data, such as temporal local variables and task queues.

- Although typically DTLB is not an issue for conventional applications, it is a significant source of inefficiencies for graph computing. In GraphBIG, a high DTLB miss penalty is observed because of the large memory footprint and low page locality.

- Graph workloads from different computation types show significant diversity in multiple architectural features. The study on graph computing should consider not only graph traversals, but also the other computation types.

- Input graph data has significant impact on memory subsystems and the overall performance. The impact is from both the data volume and the graph topology.

The major inefficiency of graph workloads comes from memory subsystem. Their extremely low cache hit rate introduces challenges as well as opportunities for future graph architecture/system research. Moreover, the low ICache miss rate of GraphBIG demonstrates the importance of proper software stack design.

3.4.3 GPU Characterization Results

We characterize the GPU workloads of GraphBIG in this section. The experiments are performed via *nvprof* on real machines. The results are summarized below.

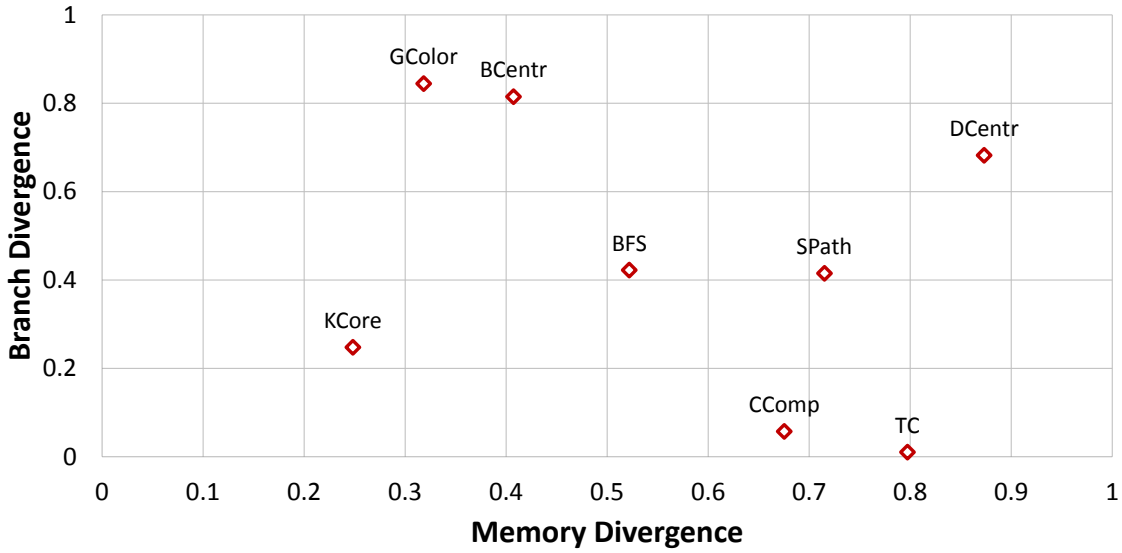


Figure 3.10: Branch and memory divergence of GraphBIG GPU workloads

Irregularity analysis: To estimate the irregularity of graph computing on GPU, we measured the degree of both branch and memory divergence. As explained in Section 3.4.1,

the metrics we used in the experiments are *branch divergence rate* (BDR) and *memory divergence rate* (MDR). With a higher BDR value, more threads in the same warp take different execution paths, leading to a lower warp utilization. Similarly, a higher MDR value indicates that more memory requests contain bank conflicts or are accessing more than 128-byte blocks. In this case, instruction replays are triggered to fulfill all memory requests.

Figure 3.10 shows a scatter plot of the workloads where the x-axis represents MDR and the y-axis represents BDR. Each dot in the figure corresponds to a GraphBIG workload. From Figure 3.10, we can observe that most workloads cannot be simply classified as branch-bound or memory-bound. A generally high divergence degree from both sides are shown. In addition, the workloads show a quite scatter distribution across the whole space. For example, kCore stays at the lower-left corner, showing relatively lower divergence in both branch and memory. On the contrary, DCentr is showing extremely high divergence in both sides. Meanwhile, branch divergence is the key issue of GColor and BCentr, while for CComp and TC, the issue is only from memory side.

The high branch divergence for graph computing comes from the thread-centric design, in which each thread processes one vertex. However, the working set size of each vertex is corresponding its degree, which can vary greatly. The unbalanced per-thread workload introduces divergent branches, especially for the loop condition checks, leading to branch divergence behaviors.In Figure 3.10, a relatively higher BDR is observed in GColor and BCentr because of the heavier per-edge computation. On the contrary, CComp and TC show small BDR values because they are both following an edge-centric model, in which each thread processes one edge.

In typical graph workloads, because of the sparse distributed edges, accesses to both the neighbor list and vertex property are spreading across the whole graph. Hence, the traversal of each edge involves cache misses and divergent memory accesses. Depending on the reliance of graph properties and warp utilization, the degree of memory divergence

may vary. As shown in Figure 3.10, the MDR value can be as low as 0.25 in kCore and as high as 0.87 in DCentr.

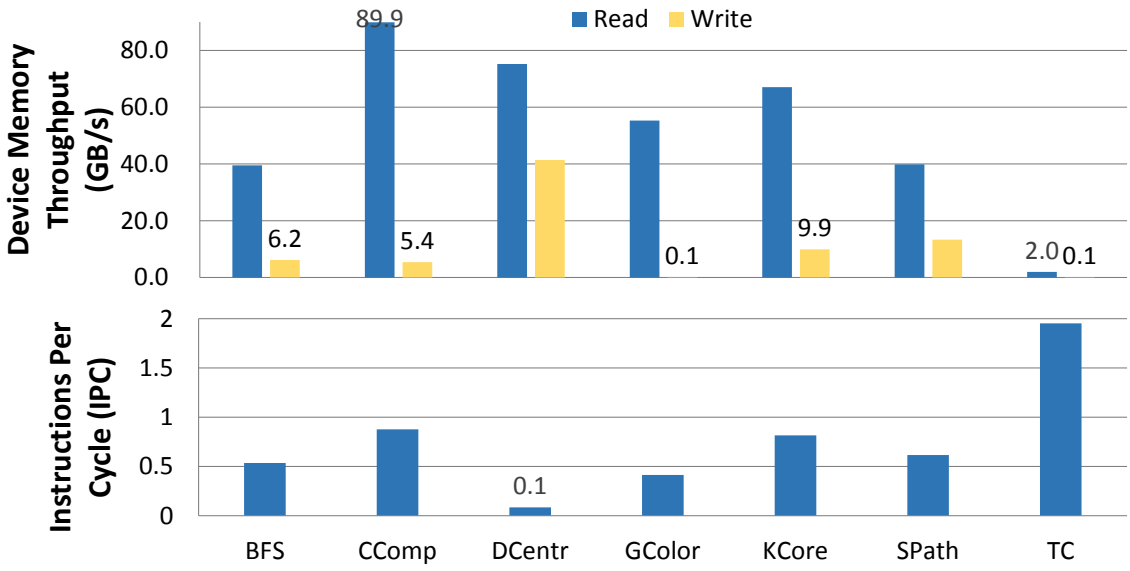


Figure 3.11: Memory throughput and IPC of GraphBIG GPU workloads

Memory throughput and IPC: The GPU device memory throughput results are shown in Figure 3.11. Although the Tesla K40 GPU supports up to 288 GB/s memory bandwidth, in our experiments, the highest read throughput is only 89.9 GB/s in CComp. The inefficient bandwidth utilization is cause by divergence in both branch and memory. The memory throughput results shown in Figure 3.11 are determined by multiple factors, including data access intensity, memory divergence, and branch divergence. For example, CComp incorporates intensive data accesses and low branch divergence. It shows the highest throughput value. DCentr has extremely intensive data accesses. Hence, even though DCentr has high branch and memory divergence, its throughput is still as high as 75.2 GB/s. A special happens at TC, which shows only 2.0 GB/s read throughput. This is because TC is mostly performing intersection operations between neighbor vertices with quite low data intensity. Since data accesses typically are the bottleneck, the memory throughput outcomes also reflect application performance in most workloads except for DCentr and TC. In DCentr, despite the high memory throughput, intensive atomic instrcutions significantly

reduce performance. Meanwhile, unlike other workloads, TC involves lots of parallel arithmetic compare operations. Hence, TC shows the highest IPC value.

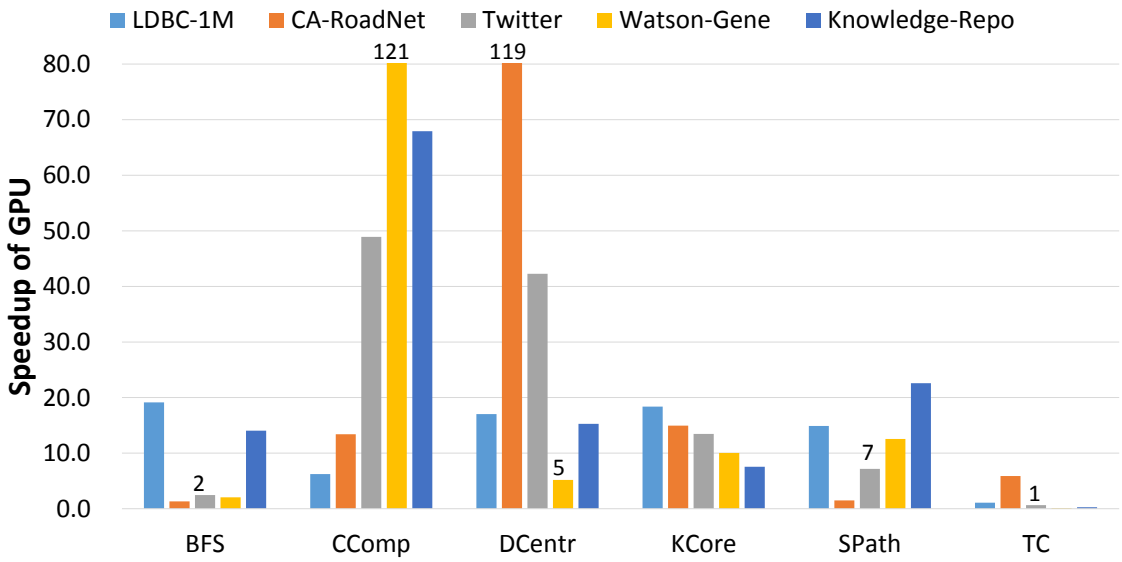


Figure 3.12: Speedup of GPU over 16-core CPU

Speedup over CPU: Although GPU is usually considered to be suitable for regular applications, irregular graph computing applications still receive performance benefits by utilizing GPUs. In Figure 3.12, the speedup of GPU over 16-core CPU is shown. In the experiments, we utilized the GraphBIG workloads that are shared between GPU and CPU sides. In our comparison, the major concern is in-core computation time, not end-to-end time. The data loading/transfer/initialize time are not included. Besides, as explained in Section 3.3, the dynamic vertex-centric data layout is utilized at CPU side, while GPU side uses CSR graph format.

From the results in Figure 3.12, we can see that GPU provides significant speedup in most workloads and datasets. The speedup can reach as high as 121 in CComp and around 20x in many other cases. In general, the significant speedup achieved by GPU is because of two major factors, thread-level parallelism (TLP) and data locality. It is difficult to benefit from instruction-level parallelism in graph applications. However, the rich TLP resources in GPUs can still provide great performance potentials. Meanwhile,

the CSR format in GPUs brings better data locality than the dynamic data layout in CPUs. Specifically, the DCentr shows high speedup number with CA-RoadNet because of the low branch divergence and static working set size. Likewise, CComp also shows similar behaviors. On the contrary, BFS and SPath show significant lower speedup values because of the low efficiency introduced by varying working set size. The speedup of TC is even lower. This is because of its special computation type. In TC, each thread incorporates heavy per-vertex computation, which is inefficient for GPU cores.

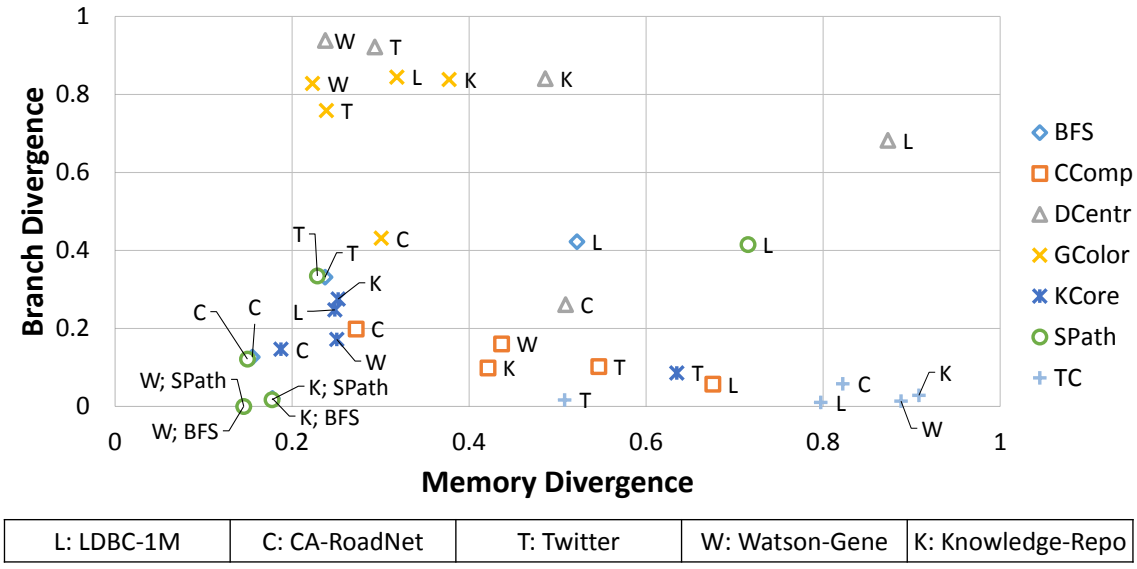


Figure 3.13: Branch and memory divergence of GraphBIG GPU workloads with different datasets

Dataset sensitivity: To estimate the sensitivity of input datasets, we performed divergence analysis on the four real-world datasets and LDBC-1M synthetic graph. In Figure 3.13, the results of different workloads are shown with different symbols in the same space, meanwhile different datasets are marked with corresponding initial letters.

From Figure 3.13, we can see that in many workloads, the divergence changes significantly for different datasets. As data-centric computing, graph workloads' behaviors are data dependent. However, the results of several datasets also tend to cluster in the same region. For CComp and TC, the branch divergence rate does not change much between different input graphs. This is expected behavior for them. They both incorporate

an edge-centric implementation, in which workload is partitioned by edges, ensuring balanced workset size between threads. Because of its low branch divergence feature, kCore also shows quite low variability in branch divergence. Both BFS and SPath show similarly low BDR values for CA-RoadNet, Watson-gene, and Knowledge-repo. This is in accordance with the graph features. Both Watson and Knowledge graphs contains small-size local sub-graphs, while CA-RoadNet includes much smaller vertex degree. Nevertheless, in Twitter and LDBC graphs, their social network features brings high BDR values. Meanwhile, unlike Twitter has a few vertices with extremely higher degree, the unbalanced degree distribution in LDBC involves more vertices. It leads to even higher warp divergence. Similar diversity happens in GColor and DCentr, which show much lower BDR values for CA-RoadNet graph because of its quite low vertex degrees.

Unlike branch divergence, MDR generally shows much higher variability for most workloads. It demonstrates the data sensitivity of memory divergence. Meanwhile, exceptions also exist. For example, BFS and SPath both show similar MDR values for CA-RoadNet, Watson-gene, and Knowledge-repo. As explained above, their special graph structures lead to a small number of traversed edges in each iteration. Thus, the impact of input graph is reduced. Moreover, the higher irregularity in edge distribution of LDBC leads to significantly higher MDR values in most workloads.

Observations

Unlike the conventional applications, the irregularity of graph computing brings unique behaviors on GPUs. We summarize the key observations as follows.

- Although graph computing is usually considered as less suitable for GPUs, with proper designs, GPU graph applications can achieve significant speedup over the corresponding CPU implementations.

- Branch divergence is known to be the top issue for graph computing on GPUs. We

observe that besides branch divergence, graph computing suffers from even higher memory divergence, leading to inefficient memory bandwidth utilizations.

- Graph workloads cannot fully utilize the GPU's execution capability. An extremely low IPC value is observed for most GraphBIG workloads.

- The behaviors of graph computing are data dependent. Input graph has comprehensive impacts on both branch and memory divergence. Specifically, memory divergence shows higher data sensitivity.

- Interestingly, the GPU graph workloads have significantly higher data sensitivity than the CPU ones. CPU/GPU sides show different data-application correlations, because of the architecture differences.

- Although traversal based workloads show similar behaviors, significant diverse behaviors across all workloads are observed. It is difficult to summarize general features that can be applied on all graph workloads. Hence, a representative study should cover not only graph traversals, but also the other workloads.

Although suffering from high branch and memory divergence, graph computing on GPUs still show significant performance benefits in most cases. Meanwhile, like CPU workloads, GPU graph computing also incorporate workload diversity and data dependent behaviors. In addition, comparing with CPU workloads, GPU graph workloads have much higher data sensitivity and more complex correlations between input data and application. To improve the performance of GPU graph computing, new architecture/system techniques considering both workload diversity and data sensitivity are needed.

3.5 Summary

In this chapter, by analyzing real-world use cases, we discussed and summarized the key factors of graph computing, including frameworks, data representations, graph computation

types, and graph data sources. We also demonstrated the impact of framework and data representation.

To understand the full-spectrum graph computing, we presented *GraphBIG*, a suite of CPU/GPU benchmarks. Our proposed benchmark suites addressed all key factors simultaneously by utilizing System G framework design and following a comprehensive workload selection procedure. With the summary of computation types and graph data sources, we selected representative workloads from key use cases to cover all computation types. In addition, we provided real-world data sets from different source types and synthetic social network data for characterization purposes.

By performing experiments on real machines, we characterized GraphBIG workloads comprehensively. From the experiments, we observed following behaviors. (1) Conventional architectures do not perform well for graph computing. Significant inefficiencies are observed in CPU memory subsystems and GPU warp/memory bandwidth utilizations. (2) Significant diverse behaviors are shown in different workloads and different computation types. Such diversity exists on both CPU and GPU sides, and involves multiple architectural features. (3) Graph computing on both of CPUs and GPUs are highly data sensitive. Input data has significant and complex impacts on multiple architecture features. As the first comprehensive architectural study on full-spectrum graph computing, GraphBIG can be served for architecture and system research of graph computing.

CHAPTER 4

GRAPHPIM: ENABLING INSTRUCTION-LEVEL PIM OFFLOADING IN GRAPH FRAMEWORKS

4.1 Introduction

As a near-data processing (NDP) technique for addressing the bottleneck in the memory subsystem, processing-in-memory (PIM) was proposed a few decades ago with a variety of proposals [24, 25, 26, 27, 28]. Unfortunately, the initial attempt for PIM was not entirely successful not only because its design and fabrication are complex but also because the problem of memory-wall was less urgent. Recently, because of advances in 3D-stacking technologies and an increasing concern about the memory bottleneck, PIM architectures have regained the attention of researchers. Several researchers have proposed various PIM architectures and programming models [29, 30, 31], and memory vendors have also started to incorporate compute units into the memory architecture as does Hybrid Memory Cube (HMC), proposed by Micron [32, 33].

In this chapter, we explore incorporating *real-world* PIM technology into graph computing to improve its execution efficiency by addressing hardware and software challenges. In particular, our study follows the HMC 2.0 specification that will be available in the near future. HMC stacks a logic layer and several DRAM layers using vertical interconnects called through-silicon vias (TSVs). The logic layer provides hardware for compute functionality and accommodates the memory controller for the DRAM layers. Starting from HMC 2.0, it supports the execution of 18 *atomic* operations in its logic layer.[1] Atomic operation support is limited to several basic operations, but it introduces the possibility of offloading computation at an an instruction granularity. To this end, we propose a full-stack

[1]HMC 2.0 differs from HMC Gen2, which follows the HMC 1.0 specification. HMC 2.0 hardware is not publicly available yet, but HMC atomic support is a practical, real-world design.

solution that enables PIM for modern graph frameworks, GraphPIM, which addresses two key challenges.

What to Offload to PIM: Exploiting PIM in an effective way requires the identification of the right candidate for offloading, which, however, has not been well discussed in prior PIM studies. GraphPIM is based on the key observation that the atomic access to the graph property is the main culprit for the inefficient execution of graph workloads on modern systems. Thus, by offloading the atomic operations on the graph property to the PIM side, GraphPIM avoids the overhead of performing the atomic operations in the host processor as well as the inefficient utilization of the memory subsystem caused by irregular data accesses.

How to Offload to PIM: Another key challenge of PIM is designing an effective interface between the host processor and PIM architecture. A recent PIM study proposed having a new host (native) instruction for every PIM operation to invoke PIM operations [31], but this may not be acceptable to some, if not most, processor vendors, because of its impact on various design aspects. Instead, GraphPIM leverages host instructions to enable PIM. The key idea of GraphPIM is to map host atomic instructions directly into PIM atomics using *uncacheable memory support* in modern architectures. With this approach, we demonstrate performance benefits for a wide range of graph workloads without any changes in ISA or user applications; requiring only a minor extension to the host processor, GraphPIM is more non-intrusive to the current software and hardware environment than other solutions, which we will discuss in Section 4.3.2.

4.2 GraphPIM Motivation

Processing-in-memory (PIM) is a decades-old concept of incorporating computation functionality directly in the memory. The integrated compute units can be fully programmable cores, such as CPU and GPU, and simple units that executes fixed-function PIM operations. As one of the first few industrial practices of PIM, Hybrid Memory Cube (HMC),

starting from HMC 2.0, provides compute capability [57]. In this section, we discuss how to exploit the PIM functionality for graph workloads.

4.2.1 Modern Graph Computing

Graph computing has been applied to a variety of domains as an important tool for processing large-scale network data. In real-world practices, because of the unique characteristics of graph data, graph computing shows distinct and diverse behaviors that are different from other computing types.

Framework-Based Computing: Unlike general applications that are often written from scratch, graph computing applications are typically implemented on top of underlying *graph frameworks* [17, 34, 9, 11], which provide user primitives for elementary graph operations, such as finding vertices and updating graph properties, and hide the complexity of graph data management from application programmers. In other words, graph frameworks decouple user application code from low-level data management and OS/hardware-related code and therefore allows us to integrate optimization techniques into the graph frameworks without adding an extra burden on programmers.

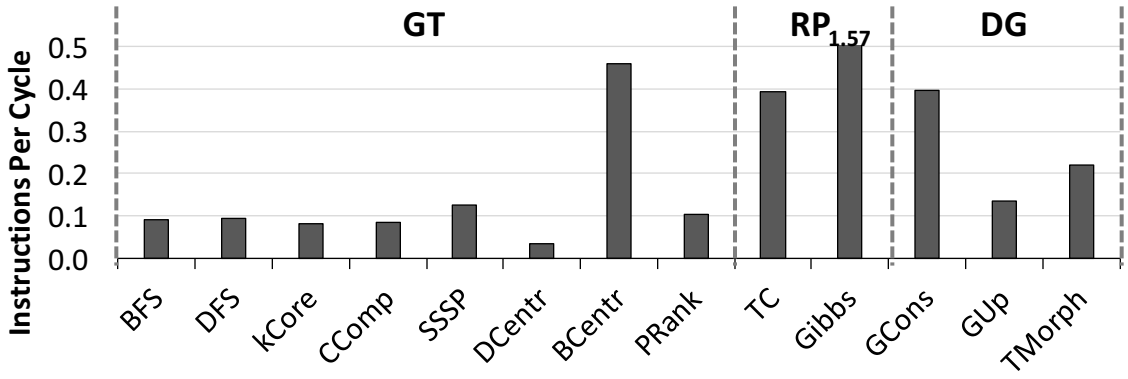


Figure 4.1: Instructions per cycle (IPC) of graph workloads on an Intel Xeon E5 machine

Inefficient Execution on Modern Systems: To understand the performance behaviors of diverse graph applications on modern architectures, we measure the instructions per cycle (IPC) of typical graph workloads in each category on an 8-core Intel Xeon E5 ma-

chine. As shown in Figure 4.1, most workloads experience extremely poor performance. For example, many applications in the GT category show IPC less than 0.1, and while the workloads in DG show a bit higher performance than GT, they are still well below IPC of 1. In general, graph computing applications (especially for the applications in the GT and DG categories) suffer from significantly poor performance on conventional architectures.

4.2.2 Bottlenecks in Graph Computing

The bottlenecks in graph computing arise from two major sources. First, graph computing typically entails a large number of *irregular memory accesses* because of the scattered graph connectivity. This makes on-chip caches mostly ineffective and thus leads to poor utilization of compute resources due to the access to the long-latency main memory. Second, graph data is typically processed in parallel due to its massive scale. Such parallel graph data processing heavily performs *atomic operations* to avoid contention of shared data. In general, atomic execution involves multiple operations and incurs non-negligible performance overhead [82]. Below, we provide an analysis to understand the bottlenecks of graph computing.

Irregular Memory Access: To understand the impact of irregular memory accesses on graph computing, we break down the execution time in the processor pipeline and measure misses per kilo instructions (MPKI) of on-chip caches across a variety of graph workloads. The experiment is performed on an Intel Xeon E5 machine using hardware performance counters [83].

First, the top graph in Figure 4.2 shows the execution time breakdown following a top-down methodology described in Intel manual [84, 83]. `Frontend` and `Backend` represent the execution time spent by frontend and backend-caused stalls.[2] Also, `BadSpeculation` shows the cycles due to miss speculation, while `Retiring` represents the cycles of successfully retired instructions. Note that the pipeline stalls caused by the memory subsystem

[2] `Frontend` includes instruction fetch, decode and allocate, and `Backend` includes instruction scheduling, executing, and retiring.

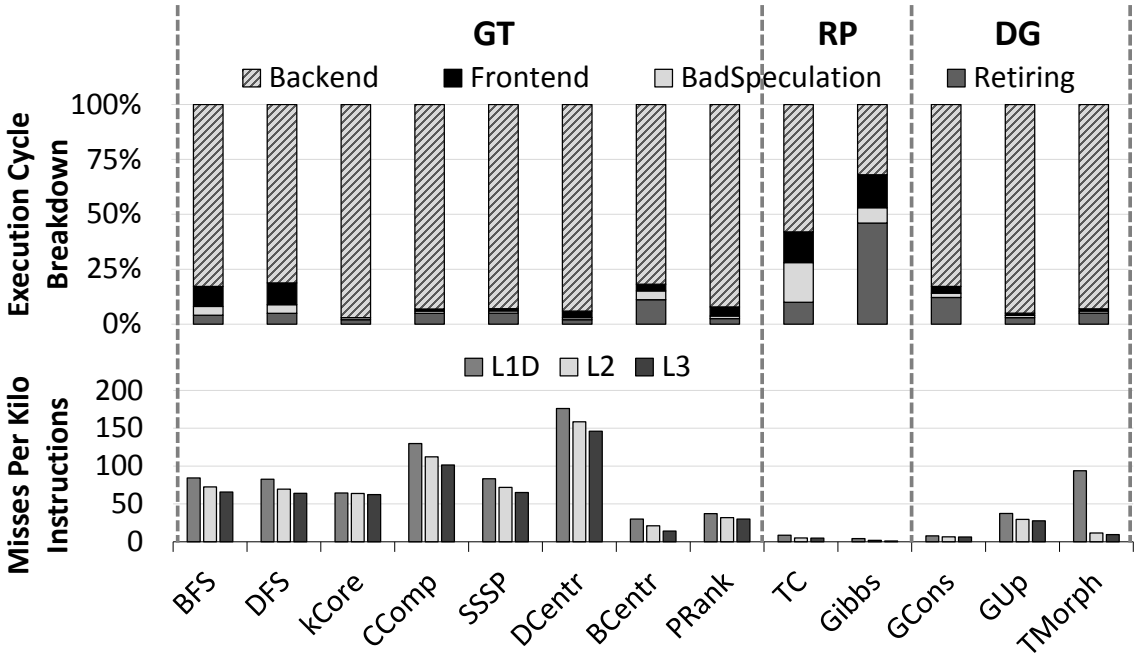


Figure 4.2: Architectural behaviors of graph workloads on an Intel Xeon E5 machine

are included in `Backend`.

As shown in the figure, graph computing spends most of its execution time on `Backend`, which is higher than 90% in some workloads, indicating that the memory subsystem is the key bottleneck for graph workloads. Such an observation is further supported by the MPKI results in the bottom graph, where L3 MPKI can be as high as 145 for the Degree Centrality (DCentr) workload. Also, the L2 and L3 caches do not provide enough benefit for most graph workloads as presented in the L2/L3 MPKI values.

Data Components: Figure 4.3 illustrates a code snippet of *breadth-first search* (BFS) using a *vertex-frontier* based algorithm [85]. The code goes through a loop that iterates over the traversal steps in a synchronized way. The algorithm in each step processes the vertices in the frontier which contains the vertices with the same depth. For each vertex, the algorithm checks the depth of its neighbors to see if it has been visited, and if not the depth information is updated using a *compare and swap (CAS)* atomic operation. Then, the newly visited vertices form the frontier for the next iteration. Here, the data access can be classified into three different components: meta data, graph structure, and graph property.

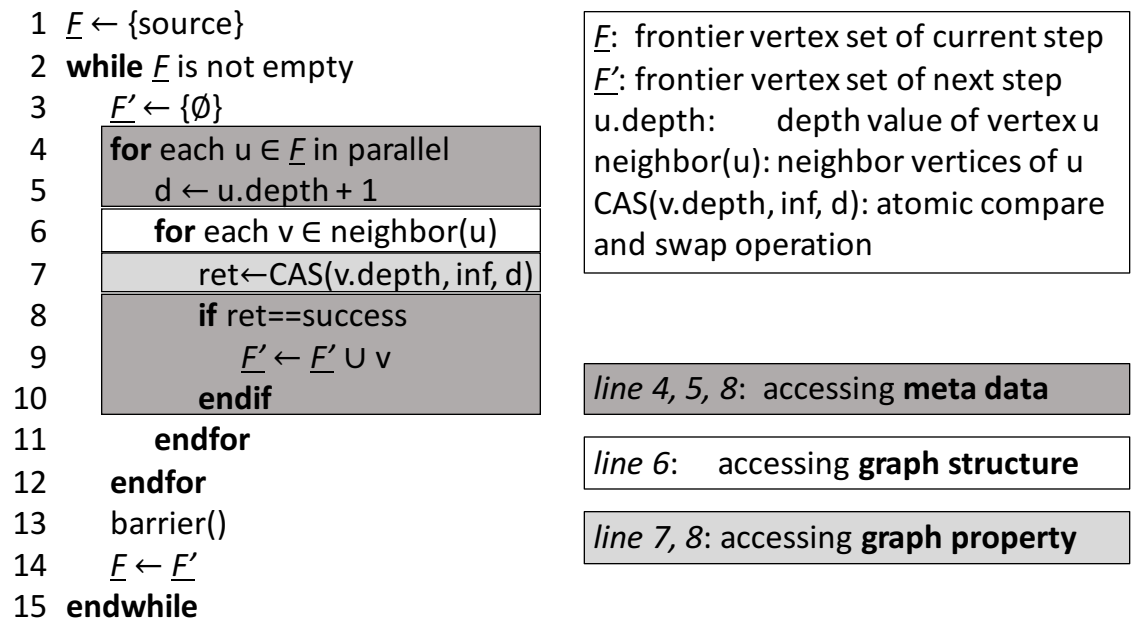


Figure 4.3: Code snippet for breadth-first search (BFS)

(1) Meta Data: Meta data include any local variables (e.g., `d`) and task queues (e.g., `F` and `F'`). These are frequently accessed and are also small in size; so, they are cache-friendly. Thus, the access to meta data mostly hits in the L1/L2 caches.

(2)Graph Structure: To check the status information of neighbors, the graph structure is accessed for retrieving the neighbor vertices. Since each vertex's neighbor list is usually organized in an array-like data structure, the access to the graph structure has good spatial locality. Thus, the memory requests to this component also do not incur a large number of main memory accesses.

(3) Graph Property: During the traversal, BFS updates the property of the neighbor vertices. Due to the irregular nature of graph connectivity, working on the property list incurs accesses that are spread throughout the entire graph. In addition, only a small portion of the graph property is captured in the cache because of the large size of graph data. As a result, the access to the graph property usually leads to a high amount of last-level cache (LLC) misses.

As explained above, the inefficient utilization of the memory subsystem is caused by

the access to the *graph property* rather than to other data components. Also, due to the uncertain nature of graph connectivity, it is challenging to improve cache performance via conventional prefetching or data remapping techniques.

In summary, we have two key observations: 1) the irregular access pattern occurs mostly in the graph property access (not spreading over all data components) and 2) the computation on the property data is a simple read-modify-write (RMW) *atomic* operation.

4.2.3 PIM Potential for Atomic Instructions

Any vertex in the graph can be shared between multiple threads. Thus, it is inevitable for most graph workloads to perform a large number of atomic operations to avoid contention on updating the shared vertex property. For example, Figure 4.3 shows that all neighbor vertices' properties are accessed via CAS atomic operations. The heavy reliance on atomic operations incurs non-negligible performance overhead in modern general-purpose architectures [82].

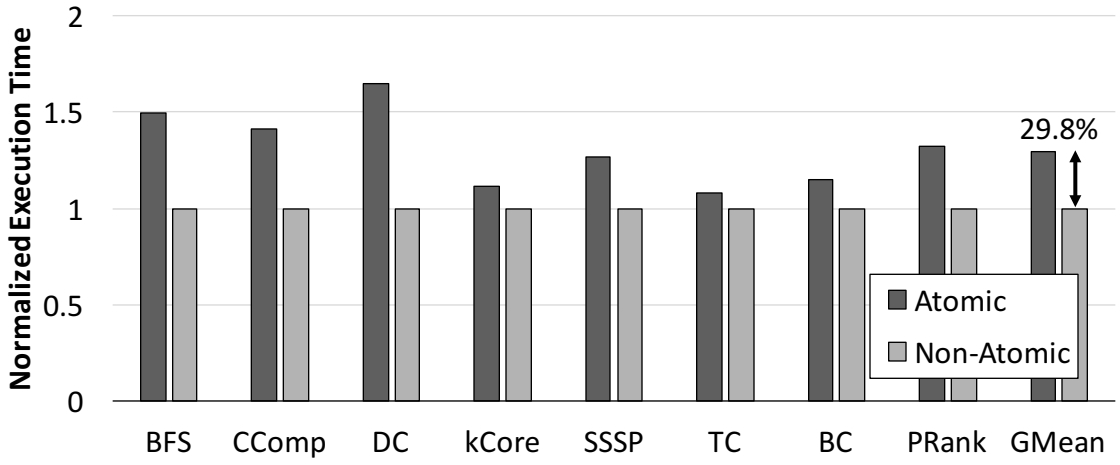


Figure 4.4: Atomic instruction overhead of graph workloads on an Intel Xeon E5 machine

To measure the overhead of atomic operations for graph workloads, we conducted an experiment on an Intel Xeon E5 machine. We created micro-benchmarks performing one iteration of each graph workload and then run the benchmarks while including/excluding the atomic operations on the graph property. As shown in Figure 4.4, compared with using

regular read and write instructions, the atomic instruction incurs a 29.8% performance degradation on average (up to 64% for `DCentr`). Such atomic overhead can potentially be avoided by utilizing the PIM offloading method, which will be further explained in the following sections.

4.3 GraphPIM Framework

4.3.1 Overview

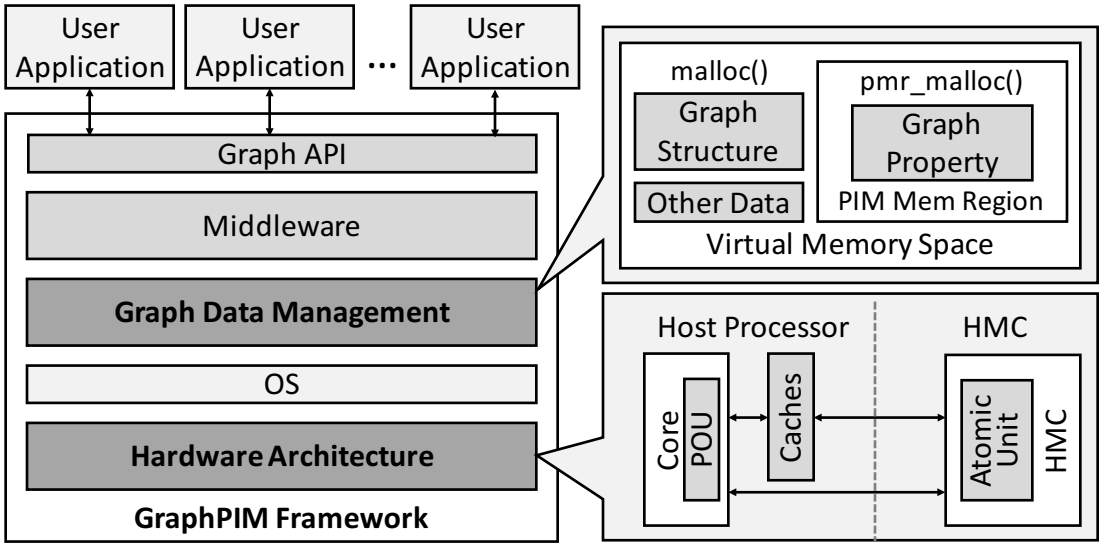


Figure 4.5: Overview of GraphPIM framework

Figure 4.5 shows an overview of our GraphPIM framework. GraphPIM enables instruction-level PIM offloading for generic graph computing frameworks with negligible changes in both software and hardware. Because of the separation of the user application layer from others, the changes are transparent to user applications.

Graph-Data Management: As explained in Section 4.2.2, both irregular memory accesses and atomic operation overhead of graph computing are caused by the accesses to the graph property. Therefore, in GraphPIM, we choose the atomic operations on the graph property as PIM offloading targets. To achieve this, GraphPIM requires the framework to allocate the graph property in the *PIM memory region (PMR)*, which is a continuous

block of uncacheable region in the virtual memory space. This is achieved by calling a customized *pmr_malloc* function (similar to jemalloc [86] and tcmalloc [87]). All host atomic instructions accessing the PMR are offloaded as PIM-Atomic requests.

Hardware Architecture: In GraphPIM, the host processor architecture implements a *PIM Offloading Unit (POU)* to determine the data path of the current memory instruction. Atomic instructions accessing the PMR will bypass the cache hierarchy and be offloaded to HMC directly. The PIM region is uncacheable, so other non-atomic memory requests to the PIM region will also bypass the cache hierarchy.

Programming Model: The application programmers can use the same graph APIs and follow the same programming model provided by the underlying graph frameworks; so, no application-level code change is required to benefit from GraphPIM from the programmer's perspective. The only change occurs *within* the framework. GraphPIM requires the graph framework to use a specialized *pmr_malloc* function to allocate memory space for graph property. This modification to the framework is negligible and does not incur extra overhead for application programmers.

4.3.2 Architectural Extensions

Figure 4.6 shows the architectural extensions to the host processor in GraphPIM. We keep the architectural changes non-intrusive to current hardware architectures.

PIM Memory Region: In GraphPIM, we define a PIM memory region (PMR) for the data of offloading targets. The PMR is specified in the virtual memory space by utilizing existing uncacheable (UC) memory support in x86 architectures [83]. The corresponding physical pages are marked as uncacheable by setting system registers (such as MTTRS in x86 [88]) from the operating systems. The underlying graph framework places the data of the offloading targets into the PMR via a customized *pmr_malloc* function at the initial memory allocation phase.

PIM Offloading Unit: In each host core, GraphPIM integrates a PIM Offloading Unit

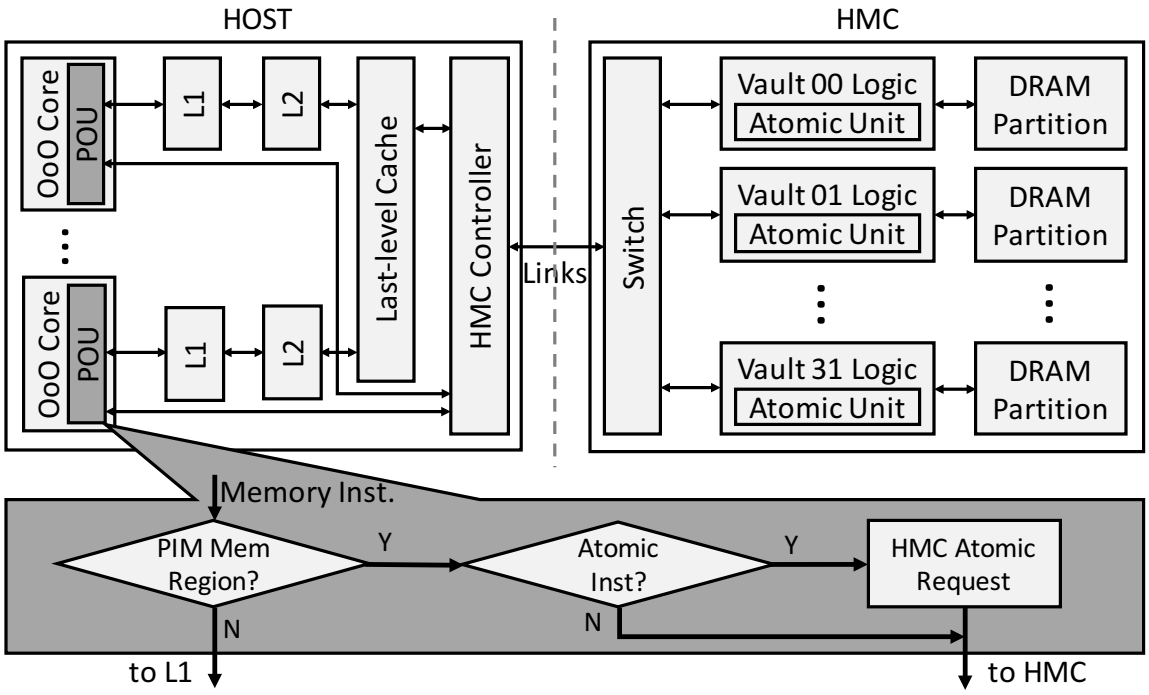


Figure 4.6: Architectural extensions for GraphPIM (added parts are shown in dark gray)

(POU), which determines the data path of memory instructions. GraphPIM does not rely on special PIM instructions in the host processor, so the host processor ISA does not need to be changed. As shown in Figure 4.6, all atomic instructions, such as instructions with a "lock" prefix in x86, are regarded as HMC operations if they are accessing the PMR. Instead of being executed in the host processor, the atomic instructions are offloaded to the HMC by sending memory requests with atomic operation commands. Note that all other non-atomic instructions bypass the cache hierarchy just as with the original UC memory support.

Cache Policy: PIM-Atomic directly modifies the data within HMC. To maintain the data coherency between HMC and cache, we follow a *cache bypassing* policy for offloading targets. By marking a page as uncacheable, all memory requests, including both offloading and non-offloading cases, will bypass the cache hierarchy if they are accessing the PMR. In this way, GraphPIM ensures that there is no data copy in the cache so that the coherence issue is avoided. Dealing with the cache bypassing policy is better than maintaining a

full coherence in terms of both performance and design complexity. Offloading targets in GraphPIM are graph property accesses, which are irregular. Thus, bypassing the caches for PIM-Atomic brings multiple benefits, such as avoiding unnecessary cache checking time, preventing cache pollution, as well as reducing memory bandwidth.

Table 4.1: Summary of PIM offloading targets

Workload	Offloading Target	PIM-Atomic Type
Breadth-first search	lock cmpxchg	CAS if equal
Degree centrality	lock addw	Signed add
Shortest path	lock cmpxchg	CAS if equal
K-core decomposition	lock subw	Signed add
Connected component	lock cmpxchg	CAS if equal
Triangle count	lock add	Signed add

Offloading Target: GraphPIM regards the host atomic instructions that access the PMR as offloading targets. Table 4.1 summarizes the offloading targets for each workload as well as the corresponding PIM-Atomic operations (the workload applicability will be further discussed in Section 4.3.3). As shown in the table, the corresponding x86 instructions with a "lock" prefix access the graph property and thus are offloaded to HMC. In the table, the graph workloads utilize two types of the PIM-Atomic operations, `CAS if equal` and `Signed add`, which can be directly mapped from the host atomic instructions. Note that there are a few PIM-Atomic operations that do not match the host atomic instructions, such as `CAS if greater` and `CAS if less`. In the host processor, the functionality of these operations is achieved via a small instruction block that consists of other existing host atomic instructions. Such an instruction block is usually generated by the compiler. To fully utilize all PIM-Atomic operations, the host architecture may incorporate a mechanism to identify such small instruction-blocks that can translate into the PIM-Atomic operations. Similar to other host atomic instructions, the identified instruction block will be regarded as a PIM offloading target if they are accessing the PMR.

Discussion: In GraphPIM, instead of adding special PIM instructions to the host processor, we choose to mark the special memory region for three major reasons: 1) Cache

coherence. When using PIM instructions, because non-offloading instructions may also access the same data, we have to maintain costly coherence between data copies in the caches and memory. Such an issue is naturally avoided in our method because of the memory address-based PIM offloading. 2) Programmer overhead. If new PIM instructions are introduced, it typically requires the effort from application programmers to modify higher level software. With our proposed method, however, there is no extra burden for application programmers using PIM. Only a simple malloc function replacement is needed in the graph framework. 3) Cache checking overhead. In our method, all data accesses to PMR will bypass the cache hierarchy. It brings extra performance benefits as explained before.

GraphPIM is utilizing the atomic operations in HMC 2.0 specification. Nevertheless, the proposed technique can be applied to other instruction-level PIM offloading environments. Likewise, GraphPIM can also be beneficial for non-graph workloads that perform atomic operations on irregular data. Besides, GraphPIM can be applied on systems equipped with both HMCs and DRAMs as well. In this case, the graph property data allocated in DRAMs will be processed in the conventional way, while the graph data in HMCs can still receive the same benefit from PIM-Atomic. In addition, it should be noted that without PIM-Atomic, it would bring huge performance degradation to bypass cache for atomic instructions because the cache-line lock will be downgraded to bus locking in this case.

4.3.3 Applicability of PIM-Atomics

As discussed in Section 4.2.1, modern graph computing covers a wide range of applications that exhibit different computation characteristics. Although we showed the feasibility of offloading the accesses to the graph property using BFS as an example, a further question may arise that whether the same technique can be applied on other graph computing applications. In this section, we discuss the applicability of PIM-Atomic operations on various graph workloads.

The current PIM-Atomic support has two major limitations. First, only simple arithmetic operations are currently implemented; complex operations (e.g., floating point operations) are not supported in HMC 2.0. Second, only one memory operand is allowed in the operations. The operations which need to specify multiple memory locations have to be split into separate requests. Thus, to benefit from GraphPIM, the target applications should fulfill two key requirements: 1) the target workloads should contain a large number of irregular memory accesses triggered by atomic operations on the graph property and 2) the atomic operations on the target memory regions should be simple enough to be mapped to the existing PIM-atomic operations. Most graph traversal applications, such as our BFS example in Figure 4.3, fulfill the requirements. To further study the applicability of PIM-Atomic on graph workloads, below we analyze all workloads from the GraphBIG benchmark suite [89].

Table 4.2: Summary of PIM-atomic applicability with GraphBIG workloads

Category	**Workload**	**Applicable? (Missing operation)**
Graph Traversal	Breadth-first search	$\surd$
	Depth-first search	$\surd$
	Degree centrality	$\surd$
	Betweenness centrality	$\times$ (Floating point add)
	Shortest path	$\surd$
	K-core decomposition	$\surd$
	Connected component	$\surd$
	Page rank	$\times$ (Floating point add)
Dynamic Graph	Graph construction	$\times$ (Complex operation)
	Graph update	$\times$ (Complex operation)
	Topology morphing	$\times$ (Complex operation)
Rich Property	Triangle count	$\surd$
	Gibbs inference	$\times$ (Computation intensive)

Inapplicable Graph Workloads: As shown in Table 4.2, most of the traversal-oriented workloads, noted as graph traversal, can make use of PIM-Atomic operations. The two exceptions are `Betweenness Centrality` and `Page Rank`, which require floating point operations. On the other hand, the graph workloads in the Dynamic Graph (DG) cate-

gory frequently perform graph structure/property updates and involve complex code structure and access patterns. Therefore, they require more complex memory operations, such as indirect accesses and multiple memory operands. For the workloads in the Rich Property (RP) category, `Triangle Count` can use PIM functionality. However, the workloads in this category perform computation within vertices' properties, so they are more computation intensive without having the irregular access pattern as other graph workloads. Thus, the PIM-Atomic may not provide performance benefits for the workload.

Potential Extension to PIM Atomics: The logic die in HMC enables the possibility of implementing a wide range of computation logic in the memory package. Although HMC 2.0 currently defines 18 simple operations, the HMC technology has the full potential of implementing new operations if needed. As previously discussed, many computing applications deal with floating point (FP) operations. For example, both `Betweenness Centrality` and `Page Rank` perfom FP add operations when updating the graph property. As floating point add/sub operations are relatively simple compared to other complex operations, FP add/sub operation support can be a reasonable extension to the future PIM-Atomic to provide PIM benefits for more graph workloads. In Section 4.4, we will further evaluate the performance benefits of supporting FP add/sub operations.

4.4 GraphPIM Evaluation

4.4.1 Methodology

We evaluate our method by using Structural Simulation Toolkit (SST) [90] with MacSim [91], a cycle-level architecture simulator. HMC is simulated by VaultSim, a 3D-stacked memory simulator based on DRAMSim2 [92].Table 4.3 shows the detailed configuration of our evaluation. We model a processor with 16 out-of-order cores and a single 8GB HMC cube that follows the HMC 2.0 specification [57, 93, 33, 94, 32]. For the workloads, we use the benchmarks from GraphBIG [89], which is a graph benchmark suite covering a wide scope of graph computing workloads. The LDBC graph is used as the input

dataset in the architectural simulation. Besides, two other large-scale graphs, bitcoin and twitter graph, are further included for evaluating our method with real-world applications.

Table 4.3: Simulation configuration

Component	Configuration
Core	16 out-of-order cores, 2GHz, 4-issue
Cache	32KB private L1 data/instruction caches 256KB private L2 inclusive cache 16MB shared L3 inclusive cache 64-byte cache line, MESI coherence protocol
HMC	8GB cube, 32 vaults, 512 DRAM banks [57] $t_{\text{CL}} = t_{\text{RCD}} = t_{\text{RP}} = 13.75\ ns,\ t_{\text{RAS}} = 27.5\ ns$ [94] 4 links per package, 120GB/s per link [57]
Benchmark Dataset	GraphBIG benchmark suite LDBC graph (1M vertex) [95], ∼900 MB footprint Bitcoin graph, ∼10 GB footprint Twitter graph, ∼5 GB footprint

4.4.2 Evaluation Results

In this section, we evaluate our proposed GraphPIM with three system configurations as explained below. All results are normalized to the baseline unless otherwise stated.

- *Baseline*: This is a conventional architecture using HMC as the main memory and does not utilize instruction offloading functionality.

- *U-PEI*: This configuration enables instruction offloading by following a mechanism similar to the previously proposed PEI [31] except that we assume perfect locality-aware offloading and ideal coherence management. In particular, all offloading requests that can hit in the cache are processed within the host processor, and the coherence between caches and HMC is assumed to incur no extra overhead. Hence, this configuration shows the performance upper-bound of PEI method.

- *GraphPIM*: This is our proposed instruction offloading technique, in which the atomic

instructions accessing the PIM memory region bypass the cache hierarchy and are offloaded to HMC.

Performance Evaluation

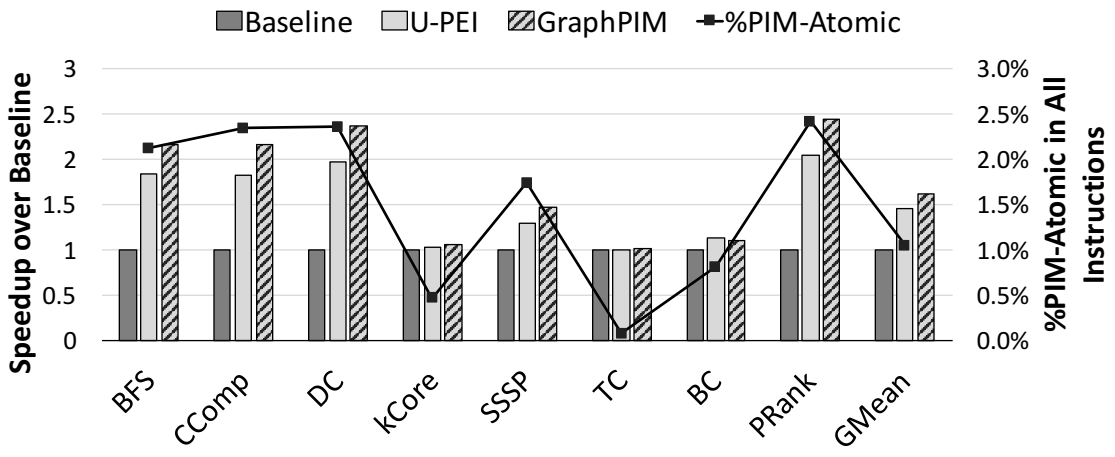


Figure 4.7: Speedups over the baseline system

Figure 4.7 shows the performance evaluation results. Compared with the baseline, GraphPIM achieves as high as 2.4× speedup (Page Rank) and more than 2× speedup for Breadth-first Search (BFS), Connected Component (CComp), and Degree Centrality (DC). On average, GraphPIM improves performance by 60% over the baseline. However, we also observe a negligible speedup for kCore Decomposition (kCore) and Triangle Count (TC). This is because they have a low percentage of offloaded PIM-Atomic operations. Thus, the performance potential is quite low to begin with. For instance, kCore spends a large amount of time on checking inactive vertices, not on accessing properties of neighbor vertices. Also, TC performs most computation within graph properties, so it is more compute intensive than other workloads.

As discussed in Section 4.3.3, with additional support for floating point add operation, Betweenness Centrality (BC) and Page Rank (PRank) can also benefit from GraphPIM. The result shows that PRank experiences a significant performance improvement with instruction offloading, while BC does not. This is because BC has a large number of centrality

computations on thread-local data structures, which makes the workload more compute-intensive and the impact of improving atomic performance relatively small.

Atomic Overhead: The major performance gain of GraphPIM is coming from avoiding the host-side atomic instructions, which incurs non-negligible overhead because of the waiting time for pending writes and extra coherence traffic.

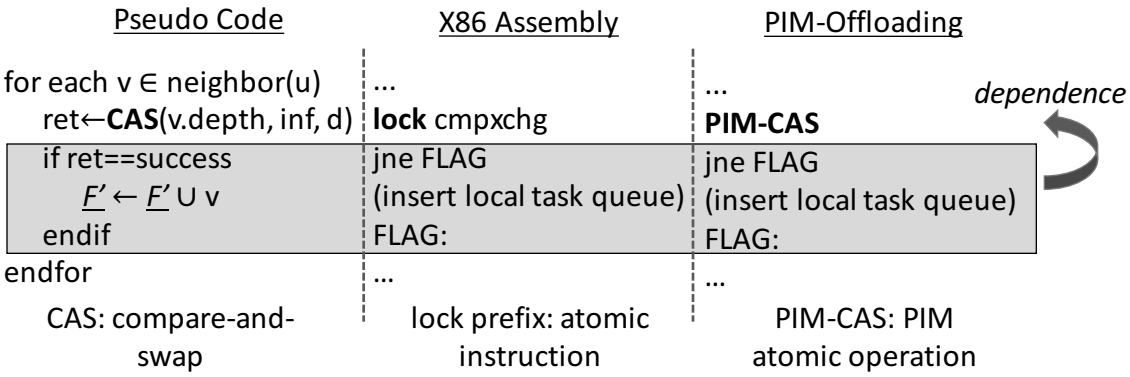


Figure 4.8: Illustration of atomic instruction overhead

In graph workloads, the long latency of atomic instructions delays not only themselves, but also following dependent instructions. As shown in Figure 4.8, the CAS operation in the pseudo code will be compiled as a lock cmpxchg instruction and then offloaded to the HMC side as a CAS-if equal operation. The following branch instruction and task queue scheduling code are depending on its return value. The long latency of atomic instructions will delay the retirement of the depending instructions. The dependent instruction block would greatly reduce the efficiency of out-of-order execution and therefore causes low processor ILP.

To estimate the impact of atomic instruction overhead, we perform experiments to measure the breakdown of atomic and non-atomic instructions in total execution time as shown in Figure 4.9. Note that atomic instructions includes not only in-core atomic overhead, but also cache checking time and coherence traffic overhead. As shown, in the baseline system, most workloads spend a large portion of their execution time in atomic instructions. For example, BFS, CComp, DC, and PR all show above 50% of atomic instruction overhead.

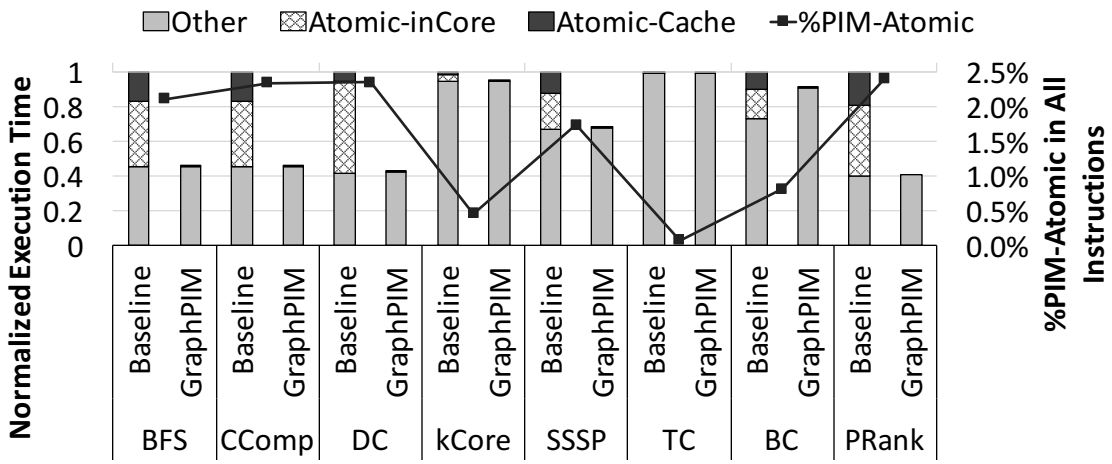


Figure 4.9: Breakdown of normalized execution time (Atomic-inCore: atomic instruction cycles for waiting pending writes; Atomic-Cache: atomic instruction cycles for cache checking and coherence traffic; Other: cycles of other instructions' execution and stall)

However, in kCore and TC, their small atomic instruction count limits the atomic overhead. Besides, in-core overhead, which includes the time for waiting pending writes, is the major source of overhead. In most workloads, above 30% of in-core overhead is observed. The result also shows a close to 20% of cache overhead. The cache overhead of atomic instructions comes from the cache checking latency for the irregular property data and extra coherence traffic.With the help of PIM offloading, GraphPIM can avoid the atomic related execution time. In GraphPIM, all workloads spend similar execution time for non-atomic part as baseline system, except for BC, in which non-atomic part requires more time. This is because in BC, non-atomic is reusing shared data from atomic part. Such data locality cannot be utilized with PIM offloading.

Cache Bypassing: We previously discussed that our offloading targets (i.e., graph property accesses) does not have data locality and therefore it is better to bypass the cache hierarchy. The experiment results also support such a conclusion. From the results in Figure 4.7, we can see that GraphPIM outperforms U-PEI because it avoids the unnecessary cache checking time. Moreover, U-PEI is already an idealized configuration, in which the extra overhead of two key factors is not included: maintaining offloaded data coherence

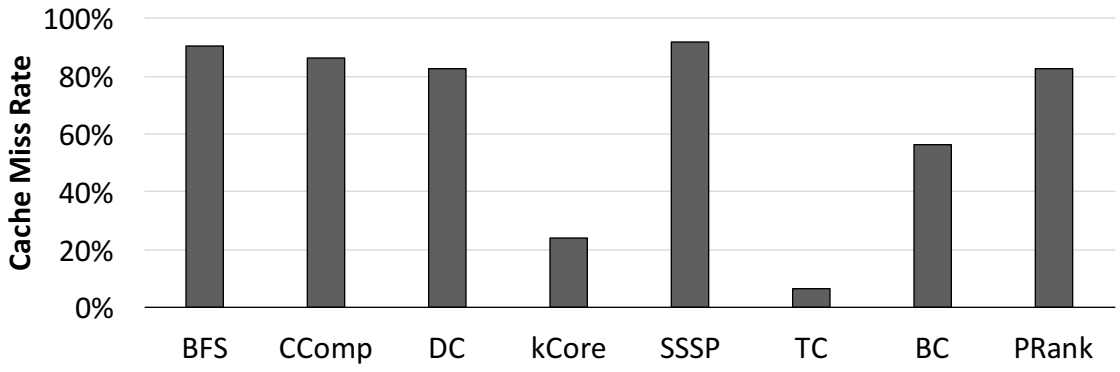


Figure 4.10: Cache miss rate of offloading candidates

and computing offloaded instructions that hit in the cache. In a more realistic system, both factors can bring significant performance overhead. Nevertheless, GraphPIM still shows a 0.2× more speedup in GMean over U-PEI. All workloads achieve better performance except BC, in which thread-local data structures are heavily used and bring data locality. In addition, a cache analysis is also performed. In most workloads, more than 80% of offloading candidates is cache miss as shown in figure 4.10, which justifies the feasibility of GraphPIM's cache policy. Furthermore, kCore, TC, and BC show relatively lower cache miss rates. However, the performance of kCore and TC is not harmed because of their limited number of accesses, whereas data locality in BC affects performance and brings slightly better speedup for U-PEI.

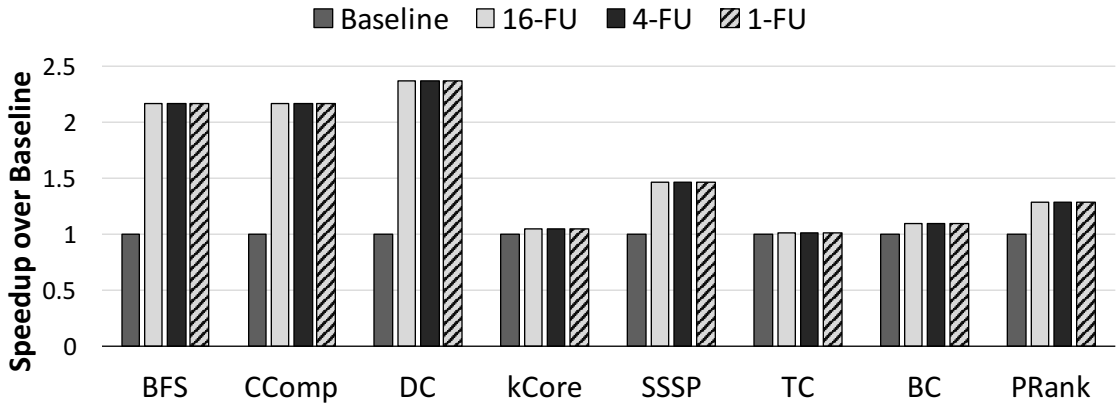


Figure 4.11: Speedup over baseline system with different functional units (FU) per HMC vault

Functional Units: An 8GB HMC contains 32 vaults, each of which has 16 memory banks. Thus, 16 functional units (FU) are enough for each vault for PIM-atomic. However, if we have fewer number of FUs in each vault, the bottleneck may be shifted to the number of FUs. To estimate the impact of the number of FUs, we perform a sensitivity analysis. As shown in Figure 4.11, there is no noticeable performance impact with different number of FUs. Even with only one FU in each vault, the performance is still roughly the same as 16-FU configuration. The result shows that the performance is not bounded by PIM-atomic throughput in HMC. This is because of two reasons: 1) HMC has 32 vaults, so the chances of consecutive HMC-Atomic requests mapped to the same vault are low. 2) The offloading target (i.e., graph property access) has depending instructions, which also introduce a large amount of interleaving memory requests. This makes PIM-atomic relatively sparse in the total memory requests. For current atomic support in HMC 2.0, the FUs consume only negligible energy even with 16 FUs per vault. However, if floating point units are incorporated, FU number can bring a substantial impact on energy consumption. We will further explain it in Section 4.4.2. In our evaluation, we assume 16 regular FUs and one floating point FU per vault.

Bandwidth Analysis

In HMC, the links between HMC and the host processor follow a packet-based protocol, where the packets consist of 128-bit flow units named as FLIT [57]. The packet size of regular memory requests and atomic operations are summarized in table 4.4. A 64-byte READ/WRITE request consumes 6 FLITs in total, while the atomic operations need only 3 or 4 FLITs. Therefore, besides the reduction in the total number of memory requests, PIM-atomic brings extra benefits due to its smaller packet size.

Figure 4.12 shows the bandwidth consumption breakdown normalized to the baseline system. We can see that GraphPIM reduces the bandwidth consumption by nearly 30% in BFS, CComp, DC, SSSP, and PRank. Because graph workloads are more intensive in

Table 4.4: HMC memory transaction bandwidth requirement in FLITs (FLIT size: 128-bit)

Type	Request	Response
64-byte READ	1 FLITs	5 FLITs
64-byte WRITE	5 FLITs	1 FLITs
add without return	2 FLITs	1 FLITs
add with return	2 FLITs	2 FLITs
boolean/bitwise/CAS	2 FLITs	2 FLITs
compare if equal	2 FLITs	1 FLITs

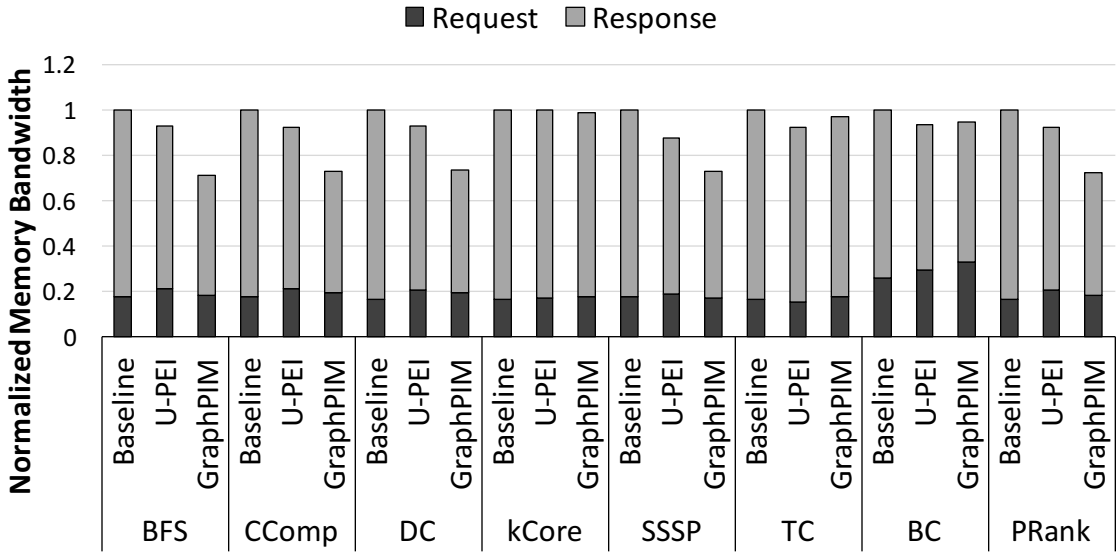


Figure 4.12: Normalized bandwidth consumption with request/response breakdown

read requests, most bandwidth savings are from the response part. Besides, the bandwidth impact of GraphPIM in kCore and TC is negligible because of their limited number of offloaded operations. Similarly, the bandwidth benefit of BC is offset by the existence of data locality. In addition, compared to U-PEI, the cache bypassing policy of GraphPIM can significantly help reduce bandwidth consumption for most workloads. For example in BFS, the bandwidth reduction is further improved from 7% to 29%. However, outliers also exist. In BC and TC, because of their data locality, GraphPIM shows slightly higher bandwidth consumption than U-PEI.

Although the reduction in memory bandwidth consumption can bring energy benefits, it is not the case in the context of performance. Figure 4.13 shows the speedup with differ-

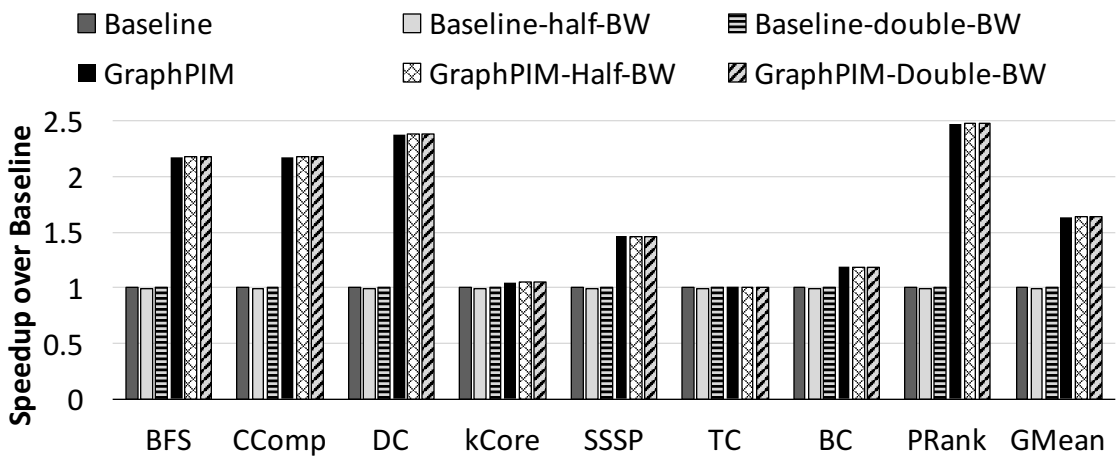


Figure 4.13: Speedup over baseline system with different HMC link bandwidth

ent HMC link bandwidth over the baseline system with original link bandwidth. As shown, since Baseline-half-BW and Baseline-double-BW are almost the same as Baseline, we can conclude that the baseline system is not sensitive to the bandwidth variations. Likewise, the speedup of GraphPIM remains the same with different HMC link bandwidth configurations. From the results, we can observe that with existing rich bandwidth resources of HMC, graph workloads are insensitive to bandwidth variations and therefore bandwidth savings cannot be effectively translated into performance gaining.

Sensitivity on Graph Size

In graph computing, input data have a significant impact on the applications' behaviors, especially for the data access pattern. To study the impact of input graph data on performance, we perform experiments using the LDBC synthetic graph [95] with four different graph sizes, from 1K vertices to 1M vertices. The dataset details are summarized in Table 4.5. The four graphs share the same social network connectivity feature but with different memory footprints.

Figure 4.14(A) shows the performance improvement of GraphPIM over U-PEI. As explained previously, the offloading target in our method is graph property access, which does not have data locality. Therefore, it is more desirable to bypass the cache hierarchy

Table 4.5: Experiment datasets

Name	Vertex #	Edge #	Footprint
LDBC-1M	1M	28.8M	∼900 MB
LDBC-100k	100K	2.8M	∼100 MB
LDBC-10k	10K	296K	∼10 MB
LDBC-1k	1K	29K	∼1 MB

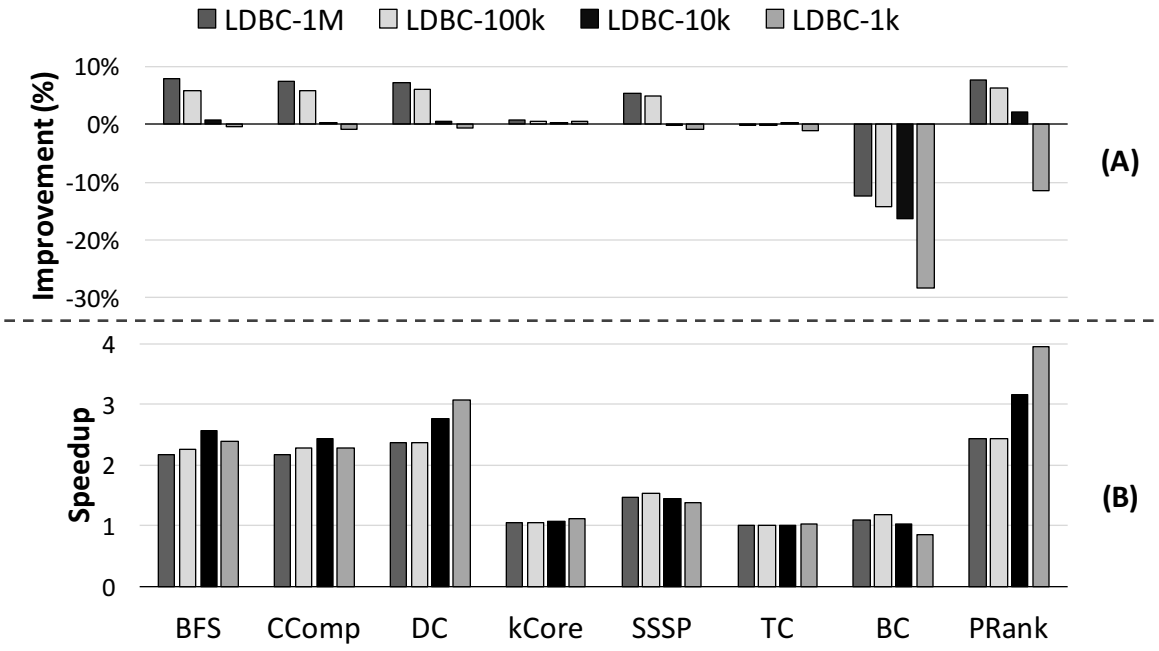


Figure 4.14: (A) GraphPIM performance improvement over U-PEI (B) GraphPIM speedup over baseline

for the offloaded operations. However, such a conclusion may be changed depending on the input data size. From the results in Figure 4.14(A), we can see that the benefit of cache bypassing decreases with smaller graph size. In some workloads, U-PEI starts to show better performance with the LDBC-10k graph. This is because the data size starts to fit into the L3 cache capacity. The degradation introduced by cache bypassing becomes much more obvious for the LDBC-1k graph. In BC, cache bypassing is always worse because of its data locality and similarly a smaller graph brings more performance degradation.

Although the benefit of cache bypassing varies with the data size, the overall performance gain of GraphPIM still remains. As shown in Figure 4.14(B), the speedup of Graph-

PIM over the baseline system does not vary as much as previous improvement results. This is because our method still can reduce significant atomic instruction overhead, which is less sensitive to the data size. Moreover, in most workloads, LDBC-10k shows a better speedup than much larger graphs. This is because the atomic instruction density of graph workloads stays at similar level with different graph sizes, while other components, such as task scheduling, are reduced with smaller graphs. Thus, sometimes smaller graphs may show even better speedup.

Energy Analysis

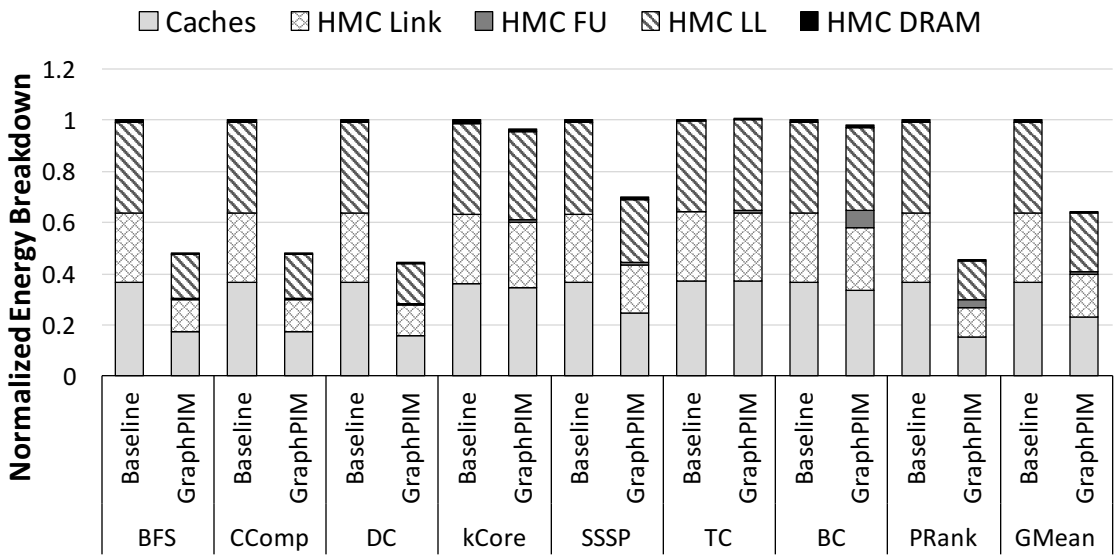


Figure 4.15: Breakdown of uncore energy consumption normalized to baseline (Caches: Host cache hierarchy; HMC Link: SerDes and data transfer; HMC FU: Functional units; HMC LL: HMC logic layer; HMC DRAM: HMC DRAM dies)

GraphPIM can save data transfer energy by reducing memory traffic. However, the PIM operation in HMC may also incur extra energy consumption. To estimate such potential trade-off, we perform analysis of uncore energy consumption in this section.

Figure 4.15 shows the uncore energy breakdown of GraphPIM normalized to the baseline. We use CACTI 6.5 [96] to model energy consumption of on-chip caches. The energy consumption of HMC SerDes links, HMC DRAM layers, and functional units in the logic

layer are computed based on the energy models from prior work [32, 97, 98]. HMC uses four high-speed Serializer/Deserializer (SerDes) links to provide high external bandwidth. However, this comes with extra energy consumption (near 43% of HMC's power [98, 32]).

As shown in the results, GraphPIM reduces the uncore energy consumption by 37% on average. The energy savings mainly come from caches, HMC links, and HMC logic layer. They are because of the reduction of cache accesses in the host side, and the reduction of memory bandwidth consumption, which saves the energy of data transfers via HMC SerDes links. Besides, GraphPIM improves performance significantly. The shorter execution time also helps reduce the uncore energy.

From the results, we can also observe that the energy consumption of HMC comes mostly from the links and logic layer. HMC FUs consume negligible energy in most workloads except for BC and PRank, in which relatively higher FU energy is shown because of floating point computation, even though GraphPIM follows a low-power design of floating point units and enables only one floating point unit per vault. As explained in previous section, the performance of graph workloads is not sensitive to FU number variations. Thus, it is more desirable to incorporate only one floating point FU per vault.

In general, GraphPIM achieves a substantial uncore energy reduction over the baseline. Even in the worst case, GraphPIM does not exceed the uncore energy consumption of the baseline. Note that beside uncore energy, we also evaluate the overall system energy and observe the same trend as the performance speedup result because of GraphPIM's significant reduction in execution time.

Real-world Applications

In real-world graph computing, large-scale graphs need to be processed with a complex combination of algorithms. To estimate the benefit of our proposed GraphPIM in real-world scenarios, we perform experiments with the following two real-world applications.

(1) Financial Fraud Detection (FD): This application is a graph-based financial fraud

detection system, which detects first-party bank fraud and money laundering behaviors. It incorporates graph traversal-based computations to uncover fraud rings in data relationships [99]. In our experiment, input data is the *Bitcoin* transaction graph [100], in which each vertex represents a Bitcoin account and each edge represents a Bitcoin transaction. The Bitcoin graph contains 71.7M vertices and 181.8M edges with around 10 GB memory footprint.

(2) Recommender System (RS): This application provides product/service recommendations for e-commerce customers. It follows an *item-to-item collaborative filtering* method [101], which is also applied in the Amazon recommender system [4]. The experiment uses a *Twitter* graph as input data [102]. It represents the friendship/followership between Twitter users. The graph contains 11M vertices and 85M edges, leading to around a 5GB memory footprint.

Table 4.6: Experiment configuration

Item	**Description**
Platform	Intel Xeon E5-2620, 2.3 GHz 2 sockets×6 cores×2 threads, 124GB memory 32KB/256KB private L1/L2, 15MB shared L3
Application	Financial fraud detection (FD) Recommender system (RS)
Dataset	Bitcoin graph, ∼10 GB memory footprint Twitter graph, ∼5 GB memory footprint

Because the application size exceeds the capability of architectural simulations, we perform real machine experiments by collecting hardware performance counters. The architectural results are then generated via an analytical model. The test platform configurations as well as application information are shown in Table 4.6.

In our analytical model, the execution cycles per instruction (CPI) is split into two components, atomic and other non-atomic instructions. Although both components may share overlapping cycles because of the out-of-order execution, the overlap is expected to

Table 4.7: Real-world application experiment results

Type	Event	FD	RS
Performance Counter	IPC	0.1	0.12
	LLC MPKI	21.3	20.6
	LLC hit rate	2.8%	13.4%
	Uncore time	65.8%	52.7%
	Backend stall	83.8%	88.8%
	%PIM-atomic	1.3%	2.9%
Analytical Model	Total host overhead	17%	32%
	Total cache checking	7%	17%

be relatively small compared to the long latency of atomic instructions. Meanwhile, in the baseline system, atomic instructions always pay the penalty of cache checking time, even though their miss rate is high. Such cache checking overhead as well as in-core atomic overhead is avoided in GraphPIM. The analyical model is summarized as follows.

$$CPI_{\text{Baseline}} = CPI_{\text{other}}(1 - overlap\%) + R_{\text{atomic}} \times (T_{\text{AO}} + Lat_{\text{cache}} + Miss_{\text{atomic}} \times Lat_{\text{mem}}) \quad (4.1)$$

$$CPI_{\text{GraphPIM}} = CPI_{\text{other}}(1 - overlap\%) + R_{\text{atomic}} \times Lat_{\text{PIM}} \quad (4.2)$$

CPI_{other} : CPI of other (non-atomic) instructions

$overlap\%$: percentage of overlapped cycles

R_{atomic} : rate of atomic instructions

T_{AO} : atomic instruction overhead

$Lat_{\text{cache}}/Lat_{\text{mem}}/Lat_{\text{PIM}}$: average cache/memory/PIM latency

$Miss_{\text{atomic}}$: miss rate of atomic instructions

Before applying the analytical model on the large-scale graph applications, we validate the correctness and accuracy of our model by comparing with previous simulation results.

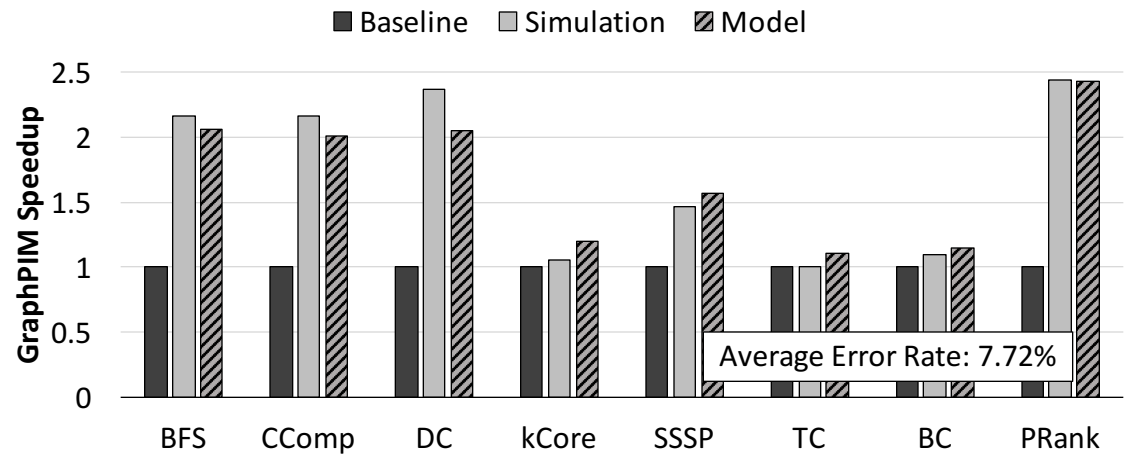


Figure 4.16: Comparison between the architectural simulation and analytical model results in speedup over baseline

As shown in Figure 4.16, our analytical model achieves a similar speedup estimation as architectural simulations. The error rate is within one-digit in most workloads, and 7.72% on average.

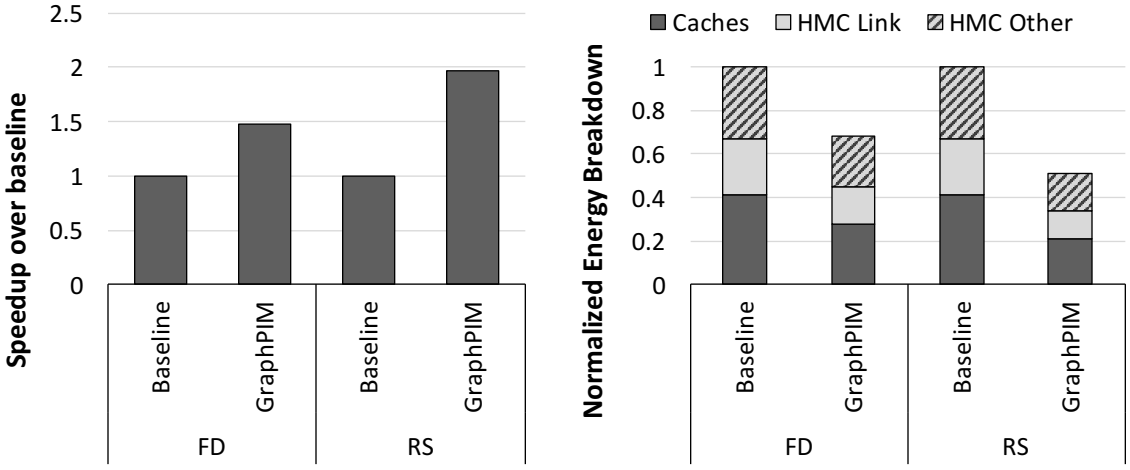


Figure 4.17: Performance and energy results of two real-world applications based on an analytical model (FD: Financial fraud detection; RS: Recommender system)

Figure 4.17 shows both performance and energy results. The energy consumption is modeled in the same way as the previous section. As shown in the results, GraphPIM significantly improves performance and energy consumption for both applications. The recommender system (RS) achieves as high as 1.9× performance speedup over the baseline.

Financial fraud detection (FD) also shows a 1.5× speedup. FD shows a bit lower performance benefit because it contains multiple non-graph computing components, which offset the overall benefit of GraphPIM. The energy comparison is also shown in Figure 4.17. GraphPIM achieves a 32% and 48% energy reduction in FD and RS respectively. The energy reduction comes from multiple factors, including cache hierarchy, data link, HMC logic, and DRAM. From the experiments of real-world applications, we can see that GraphPIM can still achieve satisfactory improvement in both performance and energy for complex real-world applications with large-scale graph data.

4.5 Discussion: Benefits of PIM Offloading

4.5.1 Bandwidth Saving

As summarized in Section 4.4, HMC follows a packet-based communication protocol. Table 4.4 summarizes the packet sizes for regular memory request and HMC-atomic instructions. PIM offloading in HMC saves bandwidth because of three main reasons: 1) unnecessary cache line data is not fetched from memory. Because the offloading targets are cache-unfriendly data, fetching the whole cache lines wastes bandwidth resources. 2) instead of issuing one load and one store instruction to the memory, we issue one HMC-atomic instruction, which consumes less bandwidth; and 3) it can further save bandwidth because we would avoid future coherence messages regarding to that cache line data.

To estimate the bandwidth savings of HMC-atomic instruction offloading, we present an analytical model for the off-chip bandwidth. Assuming 64-byte cache lines, let N_c be the number of possible offloading candidates in an application. Also, let N_{reg} be the number of regular memory requests. Assume α and β are the average LLC hit ratio ($1 - \text{LLC}_{\text{MR}}$) of offloading candidates and regular memory requests, respectively. Because each regular read/write request consumes 6 FLITs, which are 96 bytes, bandwidth usage without

offloading ($BW_{w/o}$) for the application in bytes is:

$$BW_{\text{w/o}} = \big(2N_c(1-\alpha) + N_{\text{reg}}(1-\beta)\big) \times 96 \tag{4.3}$$

In this equation, each offloading candidate contains one read and one write instruction. Therefore, we multiply N_c by two. When we perform a single HMC-atomic instruction, typically three or four FLITs are required. Therefore, assuming an equal packet size of four FLITs (4×16 bytes) for all offloading candidates, the bandwidth usage with offloading ($BW_{offloading}$) in bytes is:

$$BW_{\text{offloading}} = \big(N_c \times 64\big) + \big(N_{\text{reg}}(1-\beta) \times 96\big) \tag{4.4}$$

Therefore, bandwidth savings in bytes with offloading is as shown in Equation 4.5.

$$BW_{\text{saving}} = BW_{\text{w/o}} - BW_{\text{offloading}} = 64N_c(2-3\alpha) \tag{4.5}$$

From Equation 4.5, we can see that the cache hit ratio of offloading candidates determines if bandwidth savings occurs. Here, the turning point for bandwidth savings happens at 66% of the cache hit ratio. Although the exact cache hit ratio for different candidates is dependent on their bandwidth requirement, system cache hierarchy, and offloading target locality, the simple analytical model shows that offloading candidate locality is a key factor in the amount of bandwidth savings, and similarly for candidate offloading decision.

More available bandwidth to the memory has a positive impact on the application performance for two main reasons: 1) each memory request has shorter latency because of the reduced queuing delay; and 2) the application can issue more memory requests in a cycle. From the memory system perspective, both reasons are translated into shorter latency of memory accesses. However, from the application perspective, if the application has the capacity to issue more memory requests, but currently is limited by the low memory band-

width of hardware, providing more memory bandwidth leads to a performance improvement. This is because this type of application exploits *memory-level parallelism (MLP)*, and we group such applications as *bandwidth-sensitive* applications. However, for other applications, such as compute-intensive applications, enabling more bandwidth usually has only a small impact on the performance, and we group them as *bandwidth-insensitive* applications.

Offloading using HMC-atomic instructions provides more available bandwidth to applications, which has the same effect as increasing the system peak bandwidth. Obviously, bandwidth-sensitive applications gain a major performance benefit by exploiting the free bandwidth enabled by bandwidth savings, but this is not true for bandwidth-insensitive applications. Therefore, although by offloading irregular data, PIM saves memory bandwidth, the overall performance benefits still depend on the application behaviors. In particular, CPU applications tend to be bandwidth-insensitive applications because of the relatively lower memory bandwidth requirement, whereas GPU application tend to be bandwidth-sensitive ones because of its high parallelism and bandwidth consumption.

4.5.2 Cache-Related Benefits

By offloading the operations on the irregular data, PIM offloading also increases the cache efficiency. As explained in Section 4.3.2, the PIM offloading here follows a cache-bypass policy, in which the target PIM data is uncacheable. Therefore, performing PIM offloading brings two sides of benefits. First, because the target data is irregular and thus is less likely in the cache, enabling PIM offloading saves the unnecessary cache-checking latency. Such benefit is also demonstrated in prior evaluation results of Figure 4.9. Second, in the baseline system, the irregular offloading data consumes a significant part of cache capacity, even though they usually miss the cache. By removing the target data out of cache, PIM offloading avoids the potential cache pollution. In other words, it increases the effective cache size, and potentially can improve the cache performance of other data accesses

because of extra cache capacity. Although the cache-related benefits depend on the cache-policy of PIM offloading, in GraphPIM, the evaluation results demonstrate a non-negligible performance impact from cache side.

4.5.3 Avoiding The Overhead of Host Atomic Instructions

In ARM and x86 architectures, generic atomic instructions incur a substantial overhead because of their consistency and ILP restrictions [103, 104]. Also, the overhead of atomic instructions in AMD and NVIDIA GPU architectures exists [105, 106]. In order to understand the overhead of host atomic instructions, we conducted a real-machine experiment on an Intel Xeon E5 machine in Section 4.2.2. As shown in Figure 4.4, compared with generic atomic instructions, using regular read and write brings 30% performance speedup on average. Such an overhead can be avoided by utilizing PIM functionality and offloading the atomic instructions. Note that the overhead of atomic instructions on CPUs is typically quite significant, leading to even hundreds of penalty cycles in the worst cases; however, the atomic overhead of GPU platforms is much smaller because of the more relaxed consistency requirements.

4.5.4 Why and When PIM Works

Why PIM works: As explained in previous sections, PIM can bring performance improvement because of the benefits from three major aspects: bandwidth, cache, and atomic. However, the performance impact of each aspect depends on both the target application and the hardware platform. Bandwidth saving enables performance speedup only for bandwidth-sensitive applications. The cache benefit improves the access latency and effective cache size. It can be achieved only when the cache-bypassing policy is used for PIM offloading. By offloading the atomic instructions, PIM can avoid the substantial atomic overhead on the host architecture. Moreover, the atomic benefit is much more significant on CPU-based platforms than GPU-based platforms because of the difference in consistency models.

When PIM works: In general, the offloading target should be the operations on irregular, cache-unfriendly data because PIM gets bandwidth and cache benefits only for low cache hit ratio data. However, one exception is the atomic instructions. Because the overhead of atomic instructions on CPU can be so high that it even offsets the performance degradation from offloading cach-hit data. For example, Figure 4.14 shows that even the application has relatively higher cache hit ratio when using small datasets; it still can achieve performance improvement with PIM offloading because of the atomic benefit. Therefore, for CPU-based platforms, the atomic instruction should be the first offloading target. In this case, the major PIM benefits are coming from avoiding atomic overhead. However, on GPU-based platforms, because of the limited atomic overhead and high bandwidth utilization, bandwidth saving is the major factor instead of atomic avoidance. Thus, on GPUs, the irregular data should be the offloading targets, and PIM get major performance benefits from memory bandwidth reduction.

4.5.5 Benefit Evaluation of GPU Graph Workloads

As shown in Section 4.4, in CPU graph benchmarks, the benefits of PIM offloading mostly come from avoiding atomic instruction overhead. The prior section also discussed and explained the major benefit sources of PIM offloading for CPU- and GPU-based platforms. To further demonstrate the PIM offloading in high memory bandwidth utilization environments, we evaluate the PIM offloading with GPU platforms in this section. As shown in Figure 4.18, we perform evaluation experiments on HMC 2.0 eligible GPU workloads from GraphBIG benchmark suite. The experiments follow the same configuration as in Section 5.4 and assume no thermal impact (please refer to Chapter 5 for thermal-awareness evaluations). We select the offloading targets by following the same methodology as the CPU workloads, that is, selecting the atomic operations on graph property. In the experiments, we compare the baseline non-offloading system with the simple PIM offloading system, and show the performance speedup and bandwidth savings.

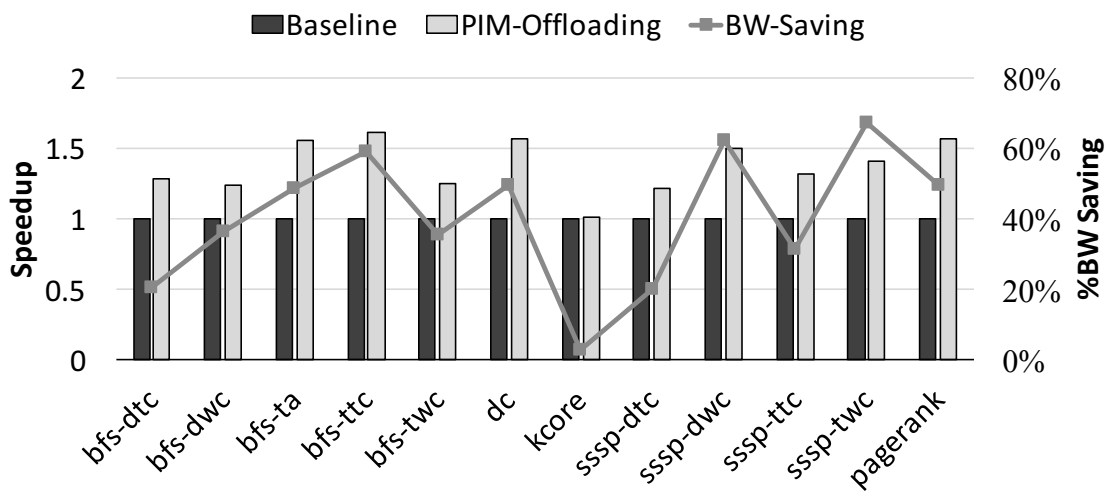


Figure 4.18: Speedup and bandwidth saving of PIM offloading for GPU graph benchmarks

From the results, we can observe that on GPU platforms, the performance gain of PIM offloading is relatively smaller than CPU workloads, achieving only up to 1.61× speedup. This is because GPUs have limited atomic overhead, and thus get benefits mainly from bandwidth savings. We can also observe that the performance gain has a strong correlation with the bandwidth savings; a higher bandwidth saving brings a higher performance gain. The exceptions exist in bfs-dtc and sssp-dtc, which have relatively low bandwidth saving percentages, but still reach 1.29× and 1.22× speedup, respectively. This is because although their total bandwidth savings are relatively small, most of their bandwidth savings happen within a few bandwidth-busy phases and help with the bandwidth congestions for the peak time. In general, the evaluation results further demonstrate our previous discussion that PIM offloading on GPU-based platforms gets performance benefits mostly from bandwidth savings because of the bandwidth-sensitive feature.

4.6 Summary

In this chapter, we present GraphPIM, a full-stack solution that enables PIM instruction offloading for graph computing. GraphPIM is built on the key observation that the atomic access to the graph property is the main culprit for the inefficient execution of graph work-

loads on modern systems. Thus, by offloading the atomic operations on the graph property to the PIM side, GraphPIM avoids the overhead of executing atomic instructions in the host processor as well as the inefficient utilization of the memory subsystem caused by irregular data accesses. Our evaluation results show that GraphPIM achieves up to a 2.4× speedup and a reduction of 37% in energy consumption for a wide range of graph benchmarks and real-world applications. GraphPIM makes use of the atomic operations specified in the HMC 2.0 specification and does not require any changes in ISA or user applications; it needs only a minor extension to the host processor and the underlying graph framework. We conclude that the technique presented in GraphPIM is a promising solution to address the bottlenecks in graph computing in a more practical way.

CHAPTER 5

COOLPIM: THERMAL-AWARE PIM OFFLOADING FOR GRAPH COMPUTING ON DATA-PARALLEL ARCHITECTURES

5.1 Introduction

Recent advances in 3D-stacking technology and the increasing need for energy efficient computing make both industry and academia to revisit near-data processing (NDP) architectures. Memory vendors such as Micron and Samsung have started productizing or publicly discussing PIM [33, 107], and recent studies from academia have also shown promising performance and energy improvements *again* [98, 30, 104] as demonstrated in the PIM research conducted decades ago [24, 108, 26, 27, 28]. With the irregular memory access behaviors of graph computing, PIM is expected to play an important role in providing extraordinary performance and energy efficiency for graph computing.

One of the key challenges to enabling 3D stacking-based PIM is to manage thermal constraints for its memory dies. For DRAM, the normal operating temperature range is under $85\,^{\circ}\mathrm{C}$.[1] Although JEDEC specifies the extended temperature range of $85\,^{\circ}\mathrm{C}$-$95\,^{\circ}\mathrm{C}$ with doubled DRAM refresh rate [109], operating DRAM in the extended temperature range incurs higher energy consumption and performance overhead than in the normal temperature range [65]. For conventional systems with dual-inline memory modules (DIMMs), the memory temperature rarely exceeds $85\,^{\circ}\mathrm{C}$. As such, DIMMs are simply used with a passive heat sink (or even without any cooling solutions) without much concern about thermal constraints or performance implications.

For the best performance of PIM, however, thermal constraints need to be more carefully taken care of because of two main reasons. First, conventional memory subsystems

[1] This is the case-surface temperature of DRAM.

only offer tens of gigabytes of memory bandwidth, but PIM implemented via 3D-stacking techniques often offers hundreds of gigabytes of memory bandwidth and raises a thermal issue when it is highly utilized as we will discuss in Section 5.2. Second, the memory dies are placed between a heat sink and a logic die in typical PIM designs, so the heat transfer capability is not as effective as in DIMMs and thus greatly increases the temperature of the memory dies. In addition, the logic layer dissipates a non-negligible amount of heat when PIM instructions are executed, which exacerbates the thermal issue to a great degree.

In this chapter, we explore managing the thermal constraints of die stacking and in-memory processing, in order to effectively utilize PIM for graph computing on data-parallel architectures that can consume hundreds of gigabytes of memory bandwidth such as GPUs [110, 111] and Xeon Phi [112]. To understand the thermal challenges, we perform an analysis on *a real HMC 1.1 prototype* while varying bandwidth utilization and cooling solutions. Our analysis shows that the HMC cannot even operate at the peak bandwidth with a *passive* heat sink, which indicates that HMC systems need a strong cooling solution even without using in-memory processing. When modeling PIM functionality as in HMC 2.0, our evaluation shows that even a commodity-server cooling could fail to maintain the memory temperature below the normal operating temperature range; so, the PIM needs to shut down for cooling down its temperature before serving memory requests again or increase/decrease the DRAM refresh rate/frequency.

Based on our observations, we propose CoolPIM, which provides both software-based and hardware-based techniques that dynamically control the intensity of PIM offloading while considering thermal conditions. The proposed technique maintains the temperature of memory dies within the normal operating temperature with a commodity-server cooling solution, which leads to higher performance compared to naïve offloading. We evaluate CoolPIM with a GPU system with a wide range of graph workloads, and our results show that CoolPIM improves performance up to 1.4x and 1.37x compared to non-offloading and naïve PIM offloading without thermal considerations.

5.2 HMC Thermal Challenges

The 3D stacking of DRAM and logic dies in HMC offers high memory bandwidth and enables PIM functionality. However, HMC exhibits non-trivial power consumption for the logic layer, especially when PIM functionality is intensively utilized. Such high power density of the logic layer increases the temperature of the stacked DRAM dies and raises the thermal challenges for the entire HMC module.

In this section, to understand the thermal challenges in HMC, we first evaluate a real HMC prototype platform while measuring the surface temperature of HMC across various bandwidth consumption and cooling methods. Then, we further evaluate an HMC 2.0 system with the energy information released in prior literature. After that, we analyze the thermal impact of PIM offloading and discuss performance trade-offs.

5.2.1 HMC Prototype Evaluation

Experiment Platform: To analyze the HMC thermal issues, we evaluate a real system that features an HMC prototype from Micron. The experiment platform (Pico SC-6 Mini [113]) has a PCIe backplane (EX-700 [114]) that can accommodate the compute modules (AC-510 [115]). Each compute module contains a Kintex Xilinx FPGA and an HMC. The HMC is a 4 GB cube and follows the HMC 1.1 specification [116]. It has two half-width (x8) links, each of which provides up to 30 GB/s of memory bandwidth. In the original system, an active heat sink is attached on top of the compute module for the cooling of both FPGA and HMC. To evaluate the thermal impact of HMC, we apply a thermal camera and measure the surface temperature of the HMC chip. The thermal resistance of a typical transistor chip is insignificant compared with an external heat sink (i.e. *plate-fin heat sink*) [117], and the in-package junction temperature should be around 5 to 10 degree higher than its surface temperature, given a 20 Watt power to dissipate.

Observations: By measuring the surface temperature of HMC with a thermal camera,

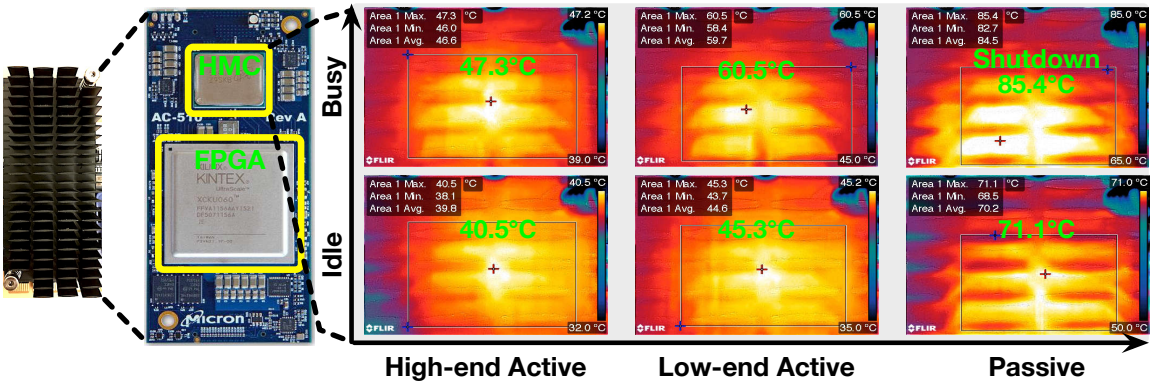


Figure 5.1: Thermal evaluation of a real HMC prototype

we evaluate the thermal impact with regard to bandwidth utilization and cooling methods. Figure 5.1 illustrates the results of our evaluation. We further elaborate our observations as follows.

1) *Surface Temperature:* HMC, unlike conventional memories, operates at a higher temperature. The runtime thermal images of the HMC under three types of heat sinks are shown in Figure 5.1. We observe that the surface temperature of highly utilized HMC exceeds 80 °C with a passive heat sink, and the junction temperature reaches to or exceeds 90 °C (with a typical thermal resistance from the package surface to the internal chip). Even with a high-end active heat sink, the surface temperature still reaches around 50 °C in our experiments. Since HMC 2.0 has a higher bandwidth than that of HMC 1.1 (60GB/s), it might experience even worse thermal issues.

2) *Overheated Behavior:* In our evaluation, with a passive heat sink, HMC cannot operate at the full bandwidth and shuts down when the surface temperature reaches around 85 °C (in-package DRAM temperature is close to 95 °C). Higher temperature makes DRAM cells weaker and charge leaking faster. Although by varying DRAM frequency and refresh interval, DRAM can operate at higher temperatures, our evaluation shows that the HMC prototype incorporates a more conservative policy, in which HMC stops completely when the DRAM stack is overheated. Although HMC can be re-enabled after the chip is cool again, the recovery delay is tens of seconds in our evaluation, which is much longer than

the processing time of typical GPU kernels.

3) *Cooling Methods:* Conventional DRAMs typically use only ambient air cooling without even a passive heat-sink, but HMC chips require much better heat transfer capability. As shown in Figure 5.1, with a passive heat sink, the surface temperature exceeds $71\,^{\circ}\mathrm{C}$ at an idle state, and HMC shuts down before the full bandwidth is achieved. Even if we include a low-end active heat sink, HMC temperature still reaches $60\,^{\circ}\mathrm{C}$ at a busy state. Thus, HMC system prefers a strong cooling mechanism. In addition, because newer generations of HMC provides substantially higher bandwidth, more efficient cooling methods are desirable.

5.2.2 Thermal Evaluation of HMC 2.0

In the previous section, we analyze the thermal issues of HMC by evaluating a real HMC 1.1 platform. However, HMC integrates PIM functionality starting from the HMC 2.0 specification [57], which has different off-chip bandwidth and architectural configurations. In this section, we conduct a thermal evaluation of HMC 2.0 without PIM instructions based on the energy data reported by Micron and derived from our gate-level synthesis.

Table 5.1: Typical cooling types

Type	**Thermal Resistance**	**Cooling Power**[b]
Passive heat sink	$4.0\,^{\circ}\mathrm{C/W}$	0
Low-end active heat sink	$2.0\,^{\circ}\mathrm{C/W}$	1x
Commodity-server active heat sink	$0.5\,^{\circ}\mathrm{C/W}$	104x
High-end active heat sink	$0.2\,^{\circ}\mathrm{C/W}$	380x[c]

[b]We follow the same plate-fin heat sink model for all configurations.
[c]The fan has 2x wheel diameter in this configuration.

Evaluation Methodology: To estimate the temperature of each layer, we perform thermal simulation using KitFox [118], a tool for integrated power and thermal simulations, and 3D-ICE [119], a 3D interlayer cooling emulator. We evaluate multiple cooling types from passive heat sink to high-end active heat sink as summarized in Table 5.1.

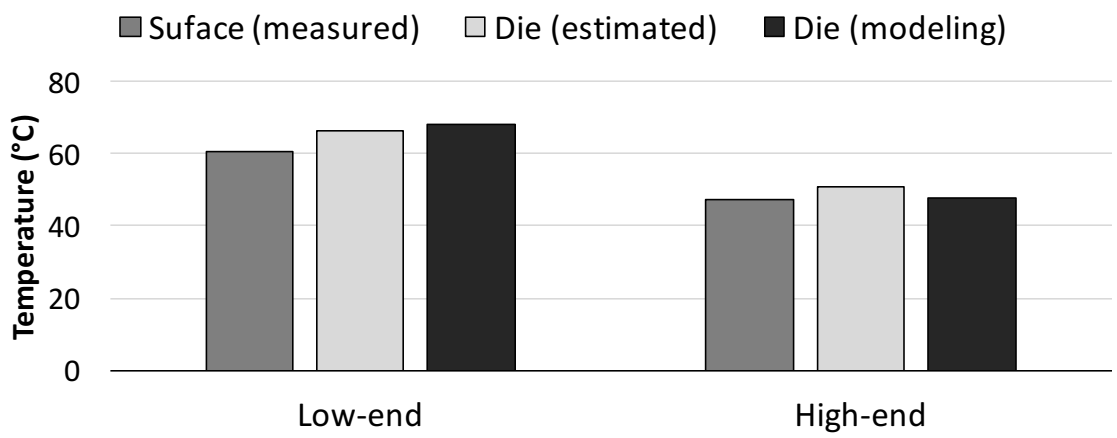


Figure 5.2: Thermal model validation. (Surface: measured surface temperature of the real HMC 1.1 chip. Die (estimated): estimated according to the surface temperature. Die (modeling): modeled die temperature)

We follow the HMC 2.0 architectural configuration of an 8GB cube, which consists of eight DRAM dies and one logic die. We assume the same energy consumption reported in the original HMC design, which are 3.7pj/bit for DRAM access and 6.78pj/bit for logic layer [32] (Please refer to Section 5.4.1 for more details about our evaluation methodology).

Model Validation: Before estimating the temperature of an HMC 2.0 cube, we first validate our thermal modeling environment by modeling an HMC 1.1 system with the same cooling and bandwidth configuration and comparing the result with previous real system measurements. Figure 5.2 shows the validation result. Our thermal modeling tool can model the temperature of DRAM dies, which is usually higher than the surface temperature we measured using a thermal camera. Thus, we also estimate the die temperature based on the surface temperature using a typical thermal resistance model. The results show that our thermal model achieves only limited error comparing with the real system temperature.

Observations: We summarize the thermal results in Figure 5.3 and Figure 5.4. With a commodity-server active heat sink, the HMC cube reaches $81\,^{\circ}\mathrm{C}$ at a full off-chip bandwidth utilization (320 GB/s). Because of the physical stacking structure of HMC, the lowest DRAM die and logic layer show the hottest temperature. From the thermal map, we can see that the hot spot appears at the center of each vault because of the high power density

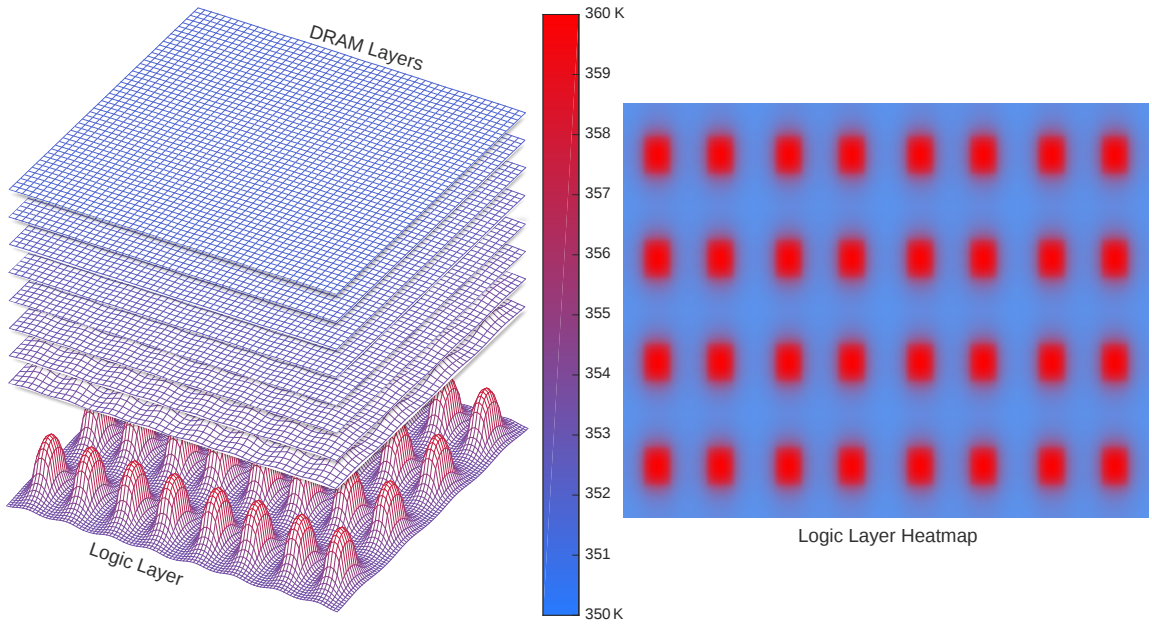


Figure 5.3: Heat map with a full bandwidth utilization and a commodity-server active heat sink (left: 3D heat map of all layers. right: 2D heat map of logic layer)

of the logic component. We also observe the thermal dependence of bandwidth utilization and cooling types summarized as follows.

1) Bandwidth Impact: The power consumption of the logic layer and DRAM dies is proportional to the bandwidth utilization. As shown in Figure 5.4, the peak DRAM temperature increases with higher bandwidth utilization. Because of the bottleneck of the off-chip link bandwidth, without PIM instructions, the maximum data bandwidth of HMC 2.0 is 320 GB/s (aggregated link bandwidth is 480 GB/s). Accordingly, with a commodity-server active heat sink, the peak DRAM temperature reaches $81\,^{\circ}\mathrm{C}$ at maximum bandwidth, and $33\,^{\circ}\mathrm{C}$ at the idle state.

2) Cooling Impact: As demonstrated in our experiments of both the real HMC prototype and the simulation, HMC temperature heavily relies on the cooling method of the HMC package. To suppress the temperature below $85\,^{\circ}\mathrm{C}$ for a full-loaded PIM, we require the thermal resistance of the cooling structure less than $0.27\,^{\circ}\mathrm{C/W}$, which falls within the realm of high-end heat sinks. However, a strong cooling method is not free. From Table 5.1, the fan power, calculated using the fan curve methodology [120], exaggerates from

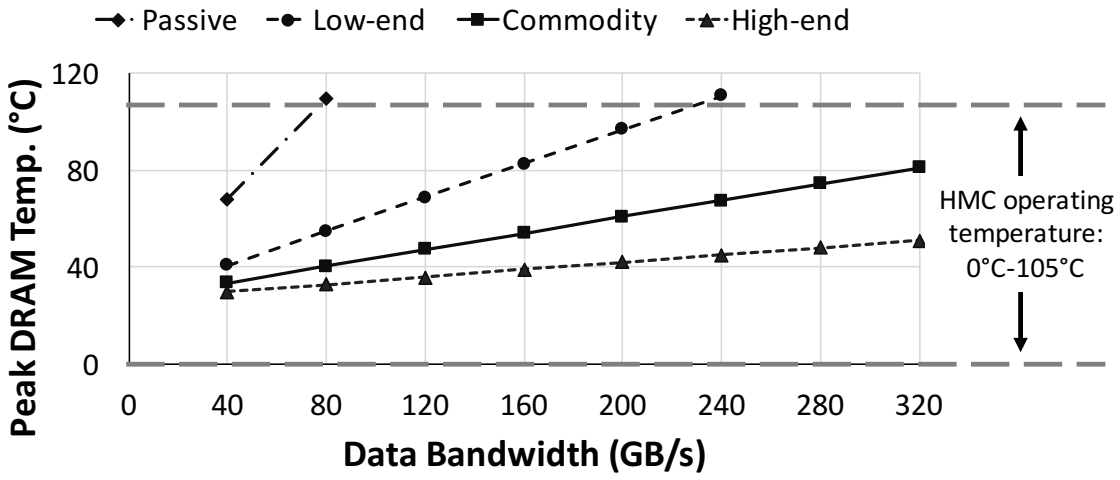


Figure 5.4: Peak DRAM temperature with various data bandwidth and cooling methods

low-end heat sinks (1x) to commodity (104x) and high-end (380x) heat sinks. Specifically, the fan in a high-end plate-fin heat sink [121] of $0.2\,^{\circ}\mathrm{C/W}$ consumes around 13 Watt (almost half as much as the power of a fully-utilized HMC 2.0 cube) in our extrapolated model of the fan power. Thus, for the sake of system power usage effectiveness (PUE), we have a limited thermal headroom for the HMC package given a restricted fan power, and the reminder of this chapter assumes commodity-servel cooling for the overall system.

5.2.3 Thermal Trade-off of PIM Offloading

As discussed in the previous section, the HMC cube maintains a relatively reasonable operating temperature with a commodity-server cooling. When PIM offloading is used, however, the HMC would experience a non-trivial temperature increment because of the additional power consumption of both logic and DRAM layers. In this section, we discuss the thermal impact of PIM offloading and its impact on performance.

Impact on Power & Temperature: For PIM functionality, the HMC cube incorporates functional units (FUs) in the logic die, and the FUs consume energy when executing offloaded PIM instructions. Hence, the additional power consumption in the logic layer is roughly proportional to the PIM offloading rate. Assuming that a single FU operation

consumes E joule/bit, the additional power consumption can be simply computed as Equation 5.1.

$$Power(FU) = E \times FU_{\text{width}} \times PIM_{\text{rate}} \tag{5.1}$$

FU_{width} is the bit width of each functional unit, which is 128bit, and PIM_{rate} is the number of PIM operations per second. To estimate the energy of a functional unit, we design a customized fixed-point ALU at a RTL level, and synthesize the design using Synopsys and a 28nm IC library [122]. The Synopsys tool reports a 1.02 pj/bit energy and 0.003 mm^2 area. Because a typical FU requires other supporting logic (e.g., FIFO and control) in addition to the ALU, which consumes roughly the similar power, we use 2.04 pj/bit for a PIM operation.

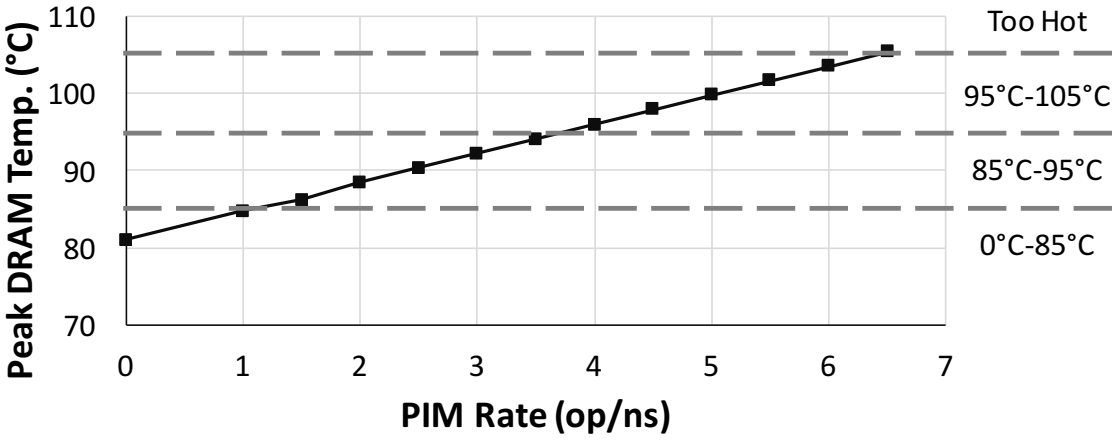


Figure 5.5: Thermal impact of PIM offloading

Each PIM instruction in HMC 2.0 requires two DRAM accesses (read and write) internally. Thus, PIM offloading increases internal DRAM bandwidth utilization, which leads to additional DRAM power consumption, which also increases the temperature. Figure 5.5 shows the relationship between the PIM offloading rate and HMC temperature. In the evaluation, we assume that a full-bandwidth utilization is achieved by the PIM operations and regular memory requests. In our modeling, maximum possible PIM offloading rate 6.5 PIM op/ns because of the thermal limitation,which reaches roughly $105\,^{\circ}\mathrm{C}$ with our FU design,

and our 3D-ICE simulation result shows a clear positive correlation between the offloading rate and temperature. Therefore, to keep the DRAM temperature below $85\,^{\circ}\mathrm{C}$, the PIM offloading rate must be lower than 1.3 operations per ns, as shown in Figure 5.5.

Performance Trade-off: By offloading operations on irregular data, a PIM system reduces external memory bandwidth consumption, which can be translated into performance benefits for bandwidth-intensive workloads. The higher offloading rate provides more bandwidth savings, and thus more performance benefits. However, as demonstrated in the prior section, the higher offloading rate also introduces the thermal problem of HMC cubes, which brings a negative impact on the performance of the memory system.

A conservative operation policy may shut down an overheated HMC completely before cooling down, thereby leading to an extremely long stall time as we observed in the HMC prototype evaluation. Moreover, if we use a dynamic DRAM management method as suggested in prior literature [63, 66], the DRAM dies in an HMC cube can also operate at a higher temperature by varying the DRAM frequency and refresh interval. However, high temperatures also can bring non-trivial performance degradation because the HMC cube will slow down not only PIM instructions, but also other regular memory requests. In typical DRAMs, starting from $85\,^{\circ}\mathrm{C}$, the operating temperature is partitioned into phases of ten degrees. For HMC 2.0 specifically, the phases of operating temperature would be $0\,^{\circ}\mathrm{C}$-$85\,^{\circ}\mathrm{C}$ (phase-1), $85\,^{\circ}\mathrm{C}$-$95\,^{\circ}\mathrm{C}$ (phase-2), and $95\,^{\circ}\mathrm{C}$-$105\,^{\circ}\mathrm{C}$ (phase-3). A higher temperature phase indicates a significant memory performance drop or even a complete shutdown. As shown in Figure 5.5, HMC reaches phase-2 temperature at 1.3 PIM offloading rate, and phase-3 temperature at 4 offloading rate. Therefore, to stay at a cool temperature for maximizing memory performance, the PIM offloading rate has be lower than 1.3. However, lower offloading rate also limits the performance benefit of PIM offloading from bandwidth savings. To strike a balance between these two factors, we propose CoolPIM, a software-based thermal-aware source throttling technique.

5.3 CoolPIM: Thermal-Aware Source Throttling

In this section, we propose CoolPIM, a simple and practical technique that controls the intensity of PIM offloading with thermal consideration. We first provide an overview of CoolPIM and then discuss our software- and hardware-based source throttling methods respectively.

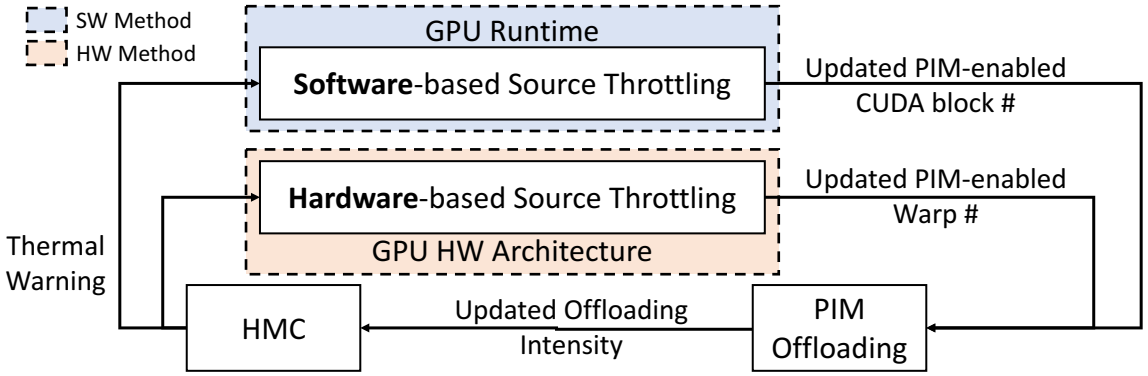


Figure 5.6: Illustration of CoolPIM feedback control

5.3.1 Overview

As shown in Figure 5.6, CoolPIM basically performs dynamic source throttling based on the *thermal warning message* from HMC. At a high level, our throttling method uses a closed-loop feedback mechanism that controls *PIM intensity*. That is, a thermal warning message leads to the reduction of the number of PIM instructions that are executed within the HMC cube, thereby decreasing the internal temperature of HMC. We present both software- and hardware-based mechanisms, which offer different throttling granularities. The software-based mechanism controls the number of *PIM-enabled CUDA blocks* launched on the GPU using a specialized thermal interrupt handler and software components in the GPU runtime; it does work without additional hardware support. In contrast, the hardware-based mechanism controls the number of *PIM-enabled warps* and thus provides a more fine-grained control, but at the cost of an extra hardware unit in each GPU

core. We explain the details of our control mechanism, starting from the software-based one, in the following sections.

5.3.2 Software-based Dynamic Throttling

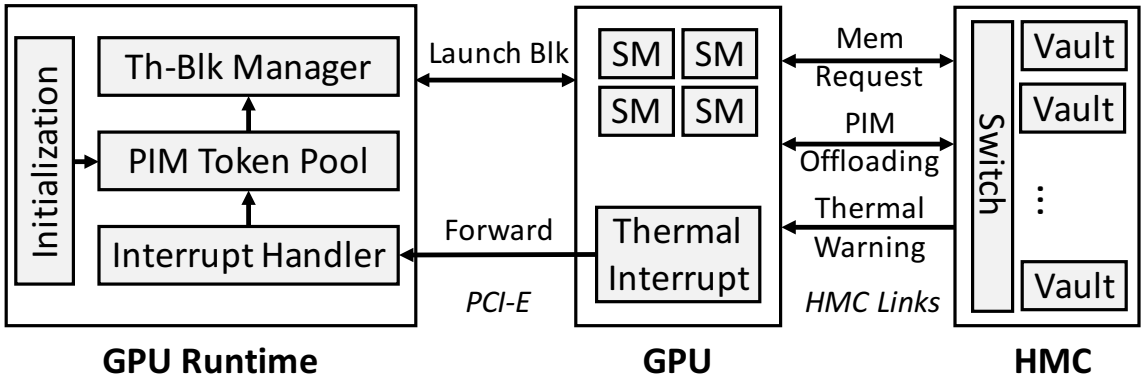


Figure 5.7: Overview of software-based dynamic throttling

Figure 5.7 shows the overview of our proposed software-based dynamic throttling technique (SW-DynT), which controls PIM offloading intensity at a *CUDA block granularity*. The GPU runtime implements an offloading controller that maintains a *PIM token pool (PTP)*. The PTP value represents the number of maximum thread blocks that are allowed to use PIM functionality. Before launching a thread block, the thread block manager first needs to request a PIM token from PTP. On a success, the runtime launches the original code that contains PIM-atomics, otherwise it launches pre-generated *shadow non-PIM code* when no token is available in the token pool. During execution, if the HMC cube gets overheated it issues a thermal warning message which will then cause SW-DynT to reduce the PTP size. Note that HMC 2.0 provides thermal feedback in its specification via error messages; therefore, no hardware modification is required.

PTP Initialization: PIM offloading intensity is controlled by updating PIM Token Pool (PTP), and the initial PTP size is determined before the execution of the target GPU applications. In fact, determining the initial size is a non-trivial issue. If the size is too large, more updates are needed to reach the proper PTP size that leads to a "cool" HMC cube.

This has an implication of a long reaction time toward the normal operating temperature, which makes HMC operate with sub-optimal performance due to the already overheated HMC. On the other hand, too small PTP sizes can also degrade performance because the dynamic control can only *down-tune* the PTP size as the thermal feedback from HMC only includes *warning messages*. Thus, the system with a too-small PTP size is likely to under-utilize available PIM resources and thermal capability. As a solution to the small PTP size, one might possibly think of increasing the PTP size step by step until receiving a warning message. In SW-DynT, however, the slow reaction time would not allow us to achieve the maximum performance opportunity.

To estimate the initial PTP size, the software-based technique performs a static analysis at compile time. As discussed in Section 5.2, the HMC temperature (HMC_{Temp}) is a function of the bandwidth utilization (BW_U) and PIM offloading rate (PIM_{Rate}), as shown in Equation 5.2.

$$HMC_{Temp} = F\big(BW_{\text{U}},\ PIM_{\text{Rate}}\big) \tag{5.2}$$

As both PIM and non-PIM code consume similar link bandwidth, the PIM offloading rate essentially dictates the difference in HMC temperatures between PIM and non-PIM code.

$$\begin{aligned} PIM_{\text{Rate}} = {} & PIM_{\text{PeakRate}} \times PIM_{\text{Intensity}} \\ & \times \left({}^{PTP\_Size}/_{\text{MaxBlk\#}}\right) \times \left(1 - Ratio_{\text{\ DivergentWarp}}\right) \end{aligned} \tag{5.3}$$

As shown in Equation 5.3, we estimate the PIM offloading rate based on PIM peak rate ($PIM_{PeakRate}$), intensity of PIM instructions ($PIM_{Intensity}$), percentage of PIM-enabled CUDA blocks, and the ratio of divergent warps ($Ratio_{\text{\ DivergentWarp}}$). Among the parameters, both PIM_{PeakRate} and MaxBlk# are hardware dependent features that can be measured by performing a simple trial run on the target platform or estimated from the hardware specification. Also, we can compute the PIM instruction intensity in the compilation stage. Although the ratio of divergent warps typically is around zero in typical GPU programs,

it can be large in some GPU applications, such as graph analytics. In such cases, we can estimate its range with the help of the algorithm knowledge. For example, topological-driven graph algorithms have a high ratio, while warp-centric ones have a low ratio. With all the parameters estimated and measured, we first estimate the required PIM offloading rate threshold from the hardware platform as Equation 5.3. Then, we compute the PTP size with the given PIM offloading rate. Because the feedback control can only down-tune the pool size, we add a small margin to the computed value in order not to be conservative (i.e. $PTP_{\text{Initial Size}} = PTP_{\text{Calculated}} + margin$); we use a margin of 4 thread blocks for our evaluation.

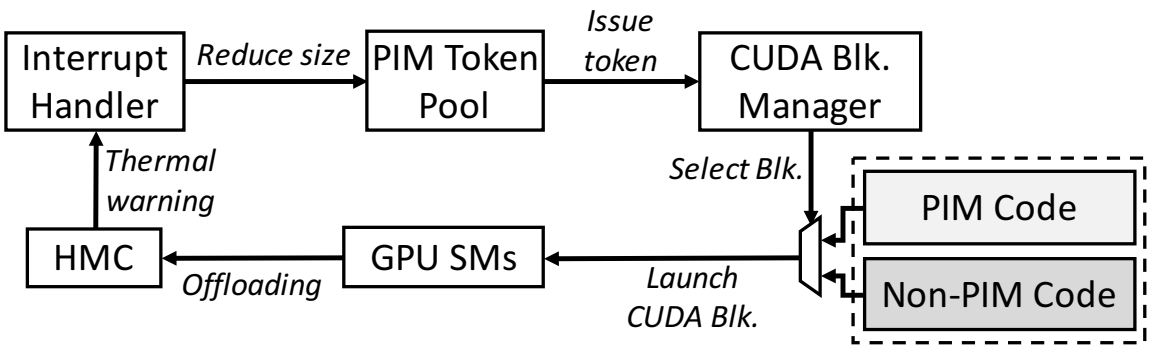


Figure 5.8: Source throttling in SW-DynT

Source Throttling: As illustrated in Figure 5.8, in our mechanism, PTP maintains the information on the maximum possible number of PIM-enabled blocks. Each new PIM-enabled CUDA block requests one token from the pool when being launched, and returns the token to the PTP when completing the execution. The PTP processes the token requests based on the first-come-first-serve policy, which issues a token to the requester until the number of on-going PIM-enabled blocks reaches the PTP size. With a successfully fetched token, the block manager launches the CUDA block using the original PIM code by configuring the corresponding code entry pointer. If the block fails to get a token, the block manager launches the block using the generated non-PIM code. The dynamic controlling of PIM and non-PIM CUDA blocks eventually determines the PIM offloading intensity and

the temperature of the HMC.

As explained in Section 2.4, when the temperature reaches a warning threshold, the HMC sets the error status bits in the response packets.[4] When receiving the thermal warning messages from the HMC side, the host GPU will trigger a thermal interrupt and forward it to the GPU runtime. The interrupt handler then updates the PTP size to reduce the offloading intensity of PIM instructions. The new PTP size will be calculated by comparing the current number of issued tokens and the PTP size after reduction as shown in Equation 5.4 where the reduction granularity is controlled by *ControlFactor (CF)*.

$$PTP_{\text{Size}} = Min\big(\ PTP_{\text{Size}} - CF,\ \#issuedToken\ \big) \tag{5.4}$$

A larger CF value allows for a fast cooldown of HMC; however, it also increases the chance of under-tuning the PTP size. On the other hand, a small control factor leads to a longer time for reducing the pool size down to the proper number and for cooling down the HMC cube. We further discuss the trade-off of different CF values later in the discussion and perform a sensitivity study in the evaluation section.

```
Void cuda_kernel(arg_list)
{
  for (int i=0; i<end; i++)
  {
    uint addr = addrArray[i];
    PIM_Add(addr, 1);
  }
}
```

Original PIM Code

```
void cuda_kernel_np(arg_list)
{
  for (int i=0; i<end; i++)
  {
    uint addr = addrArray[i];
    cuda atomicAdd(addr, 1);
  }
}
```

Shadow Non-PIM Code

Figure 5.9: Code-generation example

Code Generation for Non-PIM Code: SW-DynT launches thread blocks with a PIM-enabled kernel or a non-PIM one depending on the thermal condition. For example, if

[4]The current HMC 2.0 specification defines a single thermal error state, but it can trivially define multiple error states as multiple unused error status bits are available in the error code.

there is no token left in PTP, the runtime launches a new block with an entry point of `cuda_kernel_np` in Figure 5.9. The GPU compiler generates PIM-enabled and non-PIM code at compile time by mapping CUDA atomic functions to PIM atomics or vise versa, as shown in Figure 5.9. All PIM atomics, including the ones defined in HMC 2.0 and the extended ones proposed in [104], can be mapped to CUDA atomic functions in a straightforward way; for example in Figure 5.9, the `PIM_Add` atomic can be mapped to the CUDA atomicAdd function. Note that code generation can be trivially performed by the GPU compiler with minor changes as this is a simple source-to-source translation at the abstract syntax tree (AST) level (or the mapping can also be done at the IR level, which is also simple to perform).

5.3.3 Hardware-based Dynamic Throttling

In addition to the software-based technique, we also present a hardware-based dynamic throttling (HW-DynT) method. By controlling PIM offloading dynamically in the GPU hardware architecture, HW-DynT enables fast thermal-feedback reaction and achieves fine-grained PIM intensity control.

Hardware PIM Offloading Control: Similar to SW-DynT, HW-DynT performs offloading control based on the thermal feedback from the HMC side. However, when receiving a thermal warning message, instead of forwarding the thermal interrupt to the GPU runtime, HW-DynT directly performs source throttling in the GPU hardware. To do so, each GPU core includes an extra hardware component called *PIM Control Unit (PCU)*. PCU collects the thermal feedback from the HMC controller and reduces the number of PIM-enabled warps by *control factor (CF) in warps* when thermal warning is triggered. The PIM-disabled warps will then execute the GPU kernel code while translating PIM instructions into non-PIM ones. In HW-DynT, because of the hardware support, we can control the intensity of PIM offloading at the warp granularity, which is more fine-grained than the thread-block granularity in SW-DynT. In addition, instead of waiting for the execution

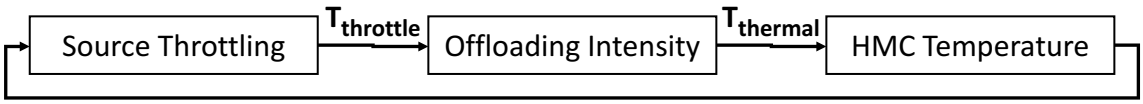
of ongoing thread-blocks to be completed, HW-DynT takes effect immediately because PIM-enabled warps can be disabled right away. Thus, because of the fast reaction of the hardware-based mechanism, HW-DynT does not require a precise initial configuration and thus just set the number of PIM-enabled warps to be maximum at the beginning.

Delayed Control Updates: HW-DynT allows for faster reaction to the thermal feedback than SW-DynT. However, although PIM offloading intensity can be changed immediately by updating the PCU in HW-DynT, the actual HMC temperature change requires a longer response time, which is usually an order of milliseconds. As a result, if HW-DynT updates the PCU too frequently in the course of the temperature change, we could over-reduce the offloading intensity. Therefore, in our mechanism, we intentionally delay the PCU updates so that the number of PIM-enabled warps will be updated only after the HMC temperature has been settled accordingly. In this way, we address the over-reduction issue and also avoid the energy and performance overhead due to the frequent PCU updates.

Dynamic PIM Instruction Translation: Because all PIM instructions in HMC 2.0 and the extended instructions proposed in [104] have the corresponding CUDA instructions, Each PIM instruction can be translated to a regular CUDA instruction dynamically according to the PIM-enable/disable condition. As shown in Table 5.2, all PIM instructions have one-to-one mapping CUDA instructions, and thus can be interpreted as regular non-PIM instructions during the decoding process in frontend.

Table 5.2: Examples of PIM instruction mapping

Type	PIM instruction	Non-PIM
Arithmetic	signed add	atomicAdd
Bitwise	swap, bit write	atomicExch
Boolean	AND/OR	atomicAND/atomicOR
Comparison	CAS-equal/greater	atomicCAS/atomicMax



	Software-based	Hardware-based
$T_{throttle}$ (Source throttling delay)	~0.1ms	~0.1us
$T_{thermal}$ (Thermal delay)	~1ms	~1ms

Figure 5.10: Delay time in our feedback control

5.3.4 Discussion

Feedback-control Granularity: Our proposed methods follow a closed-loop feedback control mechanism to keep the HMC temperature within a proper range. Thermal warning messages will trigger source throttling in the corresponding software or hardware components. SW-DynT reduces the number of PIM-enabled CUDA blocks, whereas HW-DynT reduces the number of PIM-enabled warps. However, source throttling does not lead to an immediate reduction of PIM offloading intensity. As shown in Figure 5.10, a delay of $T_{throttle}$ will be introduced (different values for SW-DynT and HW-DynT). Similarly, the HMC temperature will have an extra delay $T_{thermal}$ regarding the variation of PIM offloading intensity. Therefore, the granularity of our feedback control cannot exceed the loop delay, which is $T_{throttle} + T_{thermal}$. If an application takes N control steps to achieve the proper HMC temperature, the overall control delay would be $N \times (T_{throttle} + T_{thermal})$.

Software-based vs. Hardware-based: CoolPIM presents software- and hardware-based techniques for dynamic source throttling. Although both follow similar feedback control mechanisms, they have differences in a number of aspects and introduce interesting trade-offs.

1) *Control delay:* As explained previously, our mechanism takes a delay of $T_{throttle} + T_{thermal}$ for each control step. For the hardware-based method (HW-DynT), the source throttling delay, $T_{throttle}$, would take only tens of cycles. However, the software-based method (SW-DynT) has a much longer $T_{throttle}$. This is because of the overhead of interrupt

handling and the delay for waiting the execution of ongoing CUDA blocks. Despite of the longer $T_{throttle}$ time in SW-DynT, for GPU kernels with short execution time, the thermal response time $T_{thermal}$ is still the major delay bottleneck. Thus, both HW-DynT and SW-DynT would have a similar control delay. However, for long-execution kernels, $T_{throttle}$ can be comparable or even larger than $T_{thermal}$. In this case, the per-step control delay of HW-DynT would be significantly shorter than SW-DynT.

2) *Initialization:* Because of the longer control delay, SW-DynT relies on a proper initialization for reducing the number of overall control steps. As previously explained, we initialize the PTP size by performing static analysis of target applications. The static analysis in SW-DynT is only a one-time effort, but it still introduces an extra processing step for software compiling. On the contrary, because of the fast control reaction, HW-DynT requires no special initialization but still can reach the proper PIM offloading intensity within a short amount of time.

3) *Complexity:* Our software-based technique utilizes the existing PIM hardware and requires only non-intrusive modifications of the software runtime. However, to achieve fine-grained and fast control of PIM offloading, the hardware-based technique requires an extra hardware component in each GPU SM, which introduces non-trivial modifications compared to the software-based technique.

Control Factor: As explained in the prior section and Equation 5.4, our method reduces the PTP size when a thermal exception occurs. The reduction granularity is controlled by the *Control Factor* (CF) parameter, which has impacts on two properties. First, CF determines the reaction speed of the feedback-control loop; that is, a larger CF makes the HMC cube cool in a shorter period of time. However, CF also affects the accuracy of the pool size control. The system may over-react when a large CF is applied. In such cases, because the PTP size can be reduced to a too small number, we could miss the potential performance benefit from higher offloading intensity. We perform a sensitivity study of the CF parameter in Section 5.4.

5.4 CoolPIM Evaluation

5.4.1 Evaluation Methodology

Figure 5.11 shows an overview of our evaluation infrastructure. First, we measure the temperature of a real HMC 1.0 hardware using a thermal camera. The results are used for validating our thermal modeling environment. Then, we model an HMC 2.0 system based on the specification and the power/area numbers derived from the Synopsys tools. Finally, we estimate the system performance by performing timing simulations together with our thermal models.

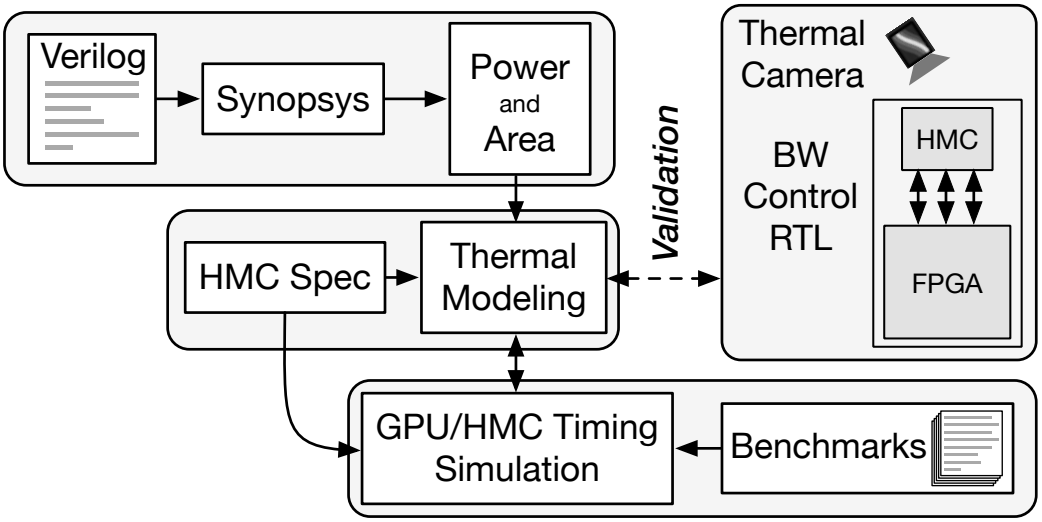


Figure 5.11: Overview of evaluation infrastructure

Power Estimation: To measure power consumption of the functional units (FUs) in HMC 2.0, we design a fixed-point functional unit in Verilog and synthesize it with Synopsys tools using a 28 nm CMOS library [122]. For DRAM and logic dies, since no public information about the design details is released, we use the energy numbers reported in prior literature from Micron [32]. Average energy consumptions per bit are 3.7 pJ/bit for DRAM layers and 6.78 pJ/bit for the logic layer. We then estimate the power consumption according to the bandwidth utilization (i.e., $power = energy/bit \times bandwidth$).

Area Estimation: HMC 1.0/1.1 die size is reported to be $68\ mm^2$ [32]. Assuming the

die is evenly partitioned into 16 vaults, each vault occupies $4.25\ mm^2$. Since no VLSI-level information of HMC 2.0 is released, we assume that HMC 2.0 occupies the same per-vault area as HMC 1.0. Similarly, we also use the same 28 nm CMOS process for the logic layer and 50 nm process for the DRAM layers. In each vault, we place a vault controller and a functional unit at the center. In our synthesis, the functional unit occupies an area of $0.003\ mm^2$. We also use the area of a vault controller synthesized in 28 nm CMOS from [123].

Thermal Modeling: To model the temperature of the HMC cube, we use KitFox [118], which is an integrated power, thermal, and reliability modeling framework. In KitFox, we use 3D-ICE [119] for a detailed thermal analysis of the 3D-stacked architecture. By following the specification, we model an 8GB HMC 2.0 which consists of a bottom logic layer and eight DRAM dies stacked on top. We assume that a commodity-server cooling capability is applied. Also, we use the same process technology, floorplan, and power consumption as previously explained in this section.

Performance Evaluation: We evaluate the performance of our proposed technique by performing detailed timing simulation because HMC 2.0 hardware is not publicly available. We use the Structural Simulation Toolkit (SST) [90] as a simulation framework, and MacSim [91], a cycle-level architecture simulator, for host-side GPU simulation. We also use VaultSim, an in-house 3D-stacked memory simulator, to model an HMC architecture. As explained in Section 5.2, for the thermal impact on HMC performance, we partition the HMC operating temperature into three phases, $0\,^{\circ}\mathrm{C}$-$85\,^{\circ}\mathrm{C}$, $85\,^{\circ}\mathrm{C}$-$95\,^{\circ}\mathrm{C}$, and $95\,^{\circ}\mathrm{C}$-$105\,^{\circ}\mathrm{C}$, and assume a 20% DRAM frequency reduction when switching to a higher temperature phase. Table 5.3 summarizes the configuration of our evaluations. We model a host GPU with 16 Streaming Multiprocessors (SMs) and an 8GB HMC cube that follows the HMC 2.0 specification. For workloads, we use the GPU benchmarks from GraphBIG [89], a comprehensive graph benchmark suite covering a wide scope of graph computing use cases, and use LDBC graph [95] (1M vertex) as the input dataset. GraphBIG also implements mul-

tiple algorithms of GPU breadth-first search (bfs) and single-source shortest path (sssp), including topology-driven/data-driven and thread-centric/warp-centric [23, 124].

Table 5.3: Performance evaluation configurations

Component	Configuration
Host	GPU, 16 PTX SMs, 32 threads/warp, 1.4GHz 16KB private L1D and 1MB 16-way L2 cache
HMC	8GB cube, 1 logic die, 8 DRAM dies 32 vaults, 512 DRAM banks [57] $t_{\text{CL}} = t_{\text{RCD}} = t_{\text{RP}} = 13.75\ ns,\ t_{\text{RAS}} = 27.5\ ns$ [94]
	4 links per package, 120GB/s per link [57] 80GB/s data bandwidth per link
	DRAM Temp. phase: 0-85 °C, 85-95 °C, 95-105 °C 20% DRAM freq reduction (high temp. phases)
Benchmark	GraphBIG benchmark suite [89] LDBC graph dataset [95]

5.4.2 Evaluation Results

In this section, we evaluate our proposed CoolPIM technique with four system configurations, which are explained as follows. In our evaluation, all speedup results are compared with the baseline system unless otherwise stated.

- *Non-Offloading (Baseline):* This is a conventional architecture using HMC as GPU memory and does not utilize PIM offloading functionality.
- *Naïve Offloading:* This configuration follows a PIM offloading mechanism similar to the previously proposed PEI [31]. PIM offloading is enabled for all thread blocks without any source control. The HMC cube uses a commodity-server active heat sink.
- *CoolPIM:* This is our proposed thermal-aware source throttling method, in which we keep the HMC cube in a cool temperature range by performing source throttling. Here, we compare both CoolPIM (SW), which is a software-based dynamic throttling

method (SW-DynT), and CoolPIM (HW), which is a hardware-based dynamic throttling method (HW-DynT). Similarly, a commodity-server active heat sink is applied on the HMC cube.

- *Ideal Thermal:* This is the scenario in which the HMC cube has an unlimited cooling capability and remains at a low temperature regardless of PIM offloading intensity.

Performance Evaluation

Figure 5.12 shows the performance results of our proposed technique. Compared with the non-offloading scenario, both software-based and hardware-based technique achieves over a 1.3× speedup for `dc`, `bfs-ta`, and `pagerank`. On average, CoolPIM improves performance by 21% for software-based method and 25% for hardware-based method over the baseline. On the contrary, instead of providing performance benefits, naïve offloading leads to performance degradation for `bfs-dwc` and `bfs-twc` by 18% and 16% respectively. Except for `sssp-dtc`, all benchmarks show only negligible or even negative performance improvements over the baseline for naïve offloading. Both `kcore` and `sssp-dtc` show the same speedup for native-offloading and CoolPIM scenarios. This is because the PIM offloading intensity is low for those workloads, which does not trigger the thermal issue of the HMC cube. In addition, we can see that in the ideal thermal scenario, PIM offloading has a great performance potential, providing a performance improvement up to 61% and 36% on average. However, such performance benefits can only be achieved with an unrealistic cooling capability, which is not applicable in a realistic system because of the non-trivial power and space overhead.

Our performance evaluation demonstrates that although PIM offloading shows a large performance benefit in the ideal thermal scenario, such performance benefits will be completely offset because of the memory slowdown triggered by the thermal issue. By controlling PIM offloading intensity from the source side, CoolPIM balances the trade-off between performance and thermal awareness.

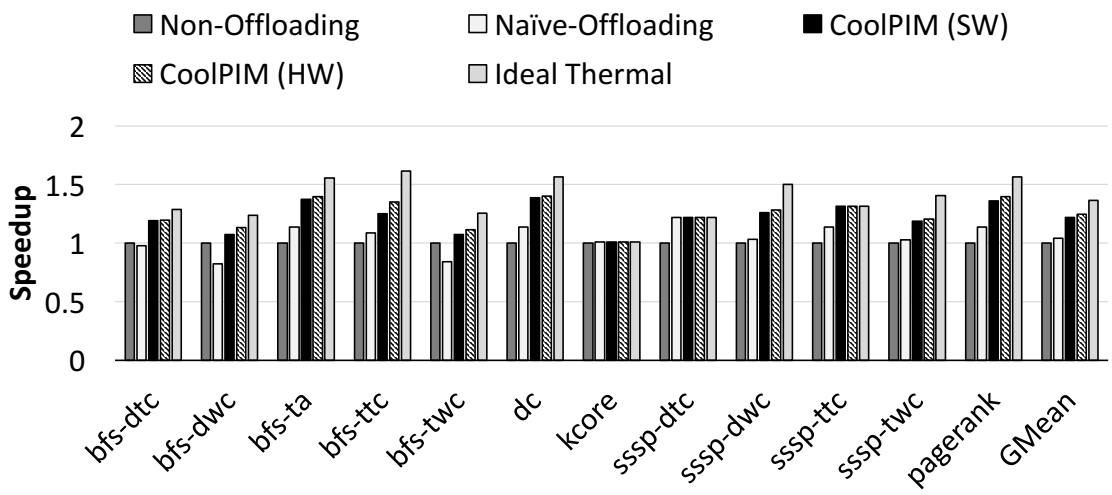


Figure 5.12: Speedup over the baseline system without PIM offloading

Bandwidth-Savings Analysis

Most of the GPU applications are bandwidth sensitive; therefore, the major source of benefits from PIM offloading is from bandwidth savings. Bandwidth savings not only improves energy efficiency of data movement, but also increases performance. Thus, it is usually an intuitive assumption that a larger bandwidth savings will lead to better performance. However, when we consider the thermal issue of PIM offloading, we observe quite different behaviors.

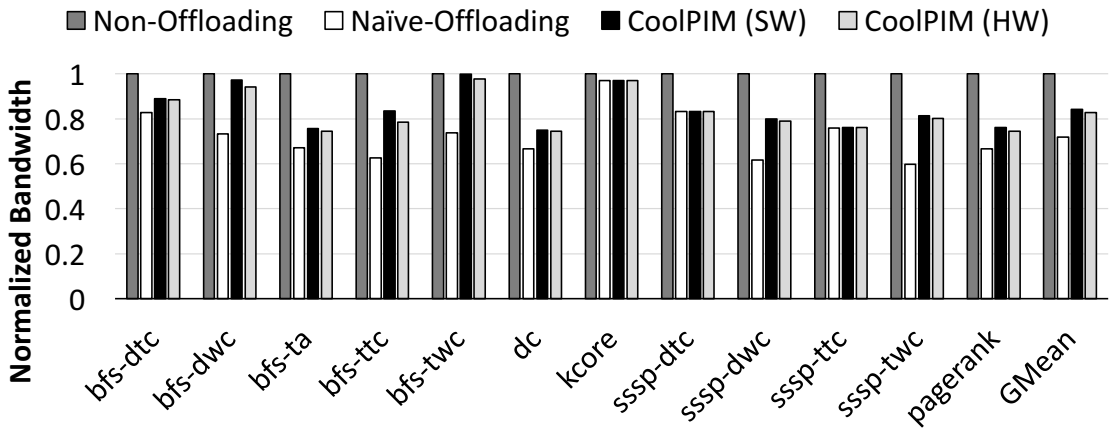


Figure 5.13: Bandwidth consumption normalized to the non-offloading baseline system

Figure 5.13 shows the bandwidth consumption of each workload normalized to the non-offloading baseline system. Naïve offloading reduces bandwidth by 39% for `sssp-dwc`, while CoolPIM (HW) only reduces bandwidth by 21% for the same benchmark. However, as shown in Figure 5.12, naïve offloading receives only a negligible performance improvement over the baseline, while CoolPIM achieves a 1.28× speedup. We can also observe the same behavior for all other benchmarks except `kcore` and `sssp-dtc`. This is because these benchmarks have a low PIM offloading intensity; therefore, offloading does not trigger a thermal issue. Figure 5.13 demonstrates that although naïve offloading enables high bandwidth savings, it instead bring performance degradation because of the thermal issues caused by the offloading.

Thermal Analysis

As explained in Section 5.2, the temperature of an HMC cube depends on the utilization of its bandwidth and intensity of PIM offloading. Because the graph computing benchmarks used in our evaluation are bandwidth saturated, the utilization of PIM offloading components between the scenarios is the main deciding factor in determining the peak DRAM temperature for the HMC cube.

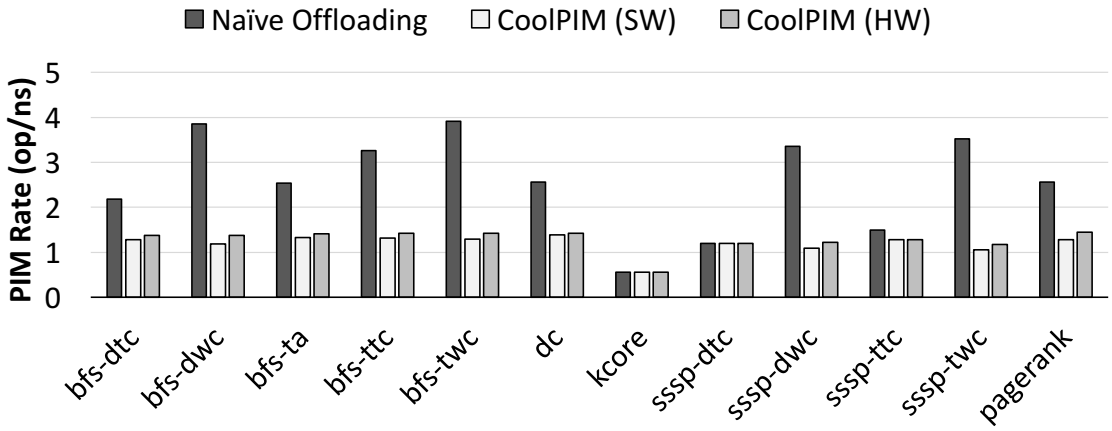


Figure 5.14: Comparison of PIM offloading rate

Figure 5.14 shows the average PIM offloading rate with a naïve-offloading method

and with our proposed CoolPIM mechanisms. Without any source throttling, naïve offloading reaches close to 4 op/ns (PIM operations per nanosecond) for `bfs-dwc` and `bfs-twc`, and more than 3 op/ns for other benchmarks, such as `bfs-ttc`, `sssp-dwc`, and `sssp-twc`. Such a high PIM offloading intensity would lead to a high temperature of the HMC cube and trigger the thermal issue. However, source throttling of CoolPIM keeps the PIM offloading rate below 1.3 op/ns for all benchmarks. The results demonstrate the effectiveness of our proposed method, which successfully keeps the PIM offloading intensity within a desirable range.

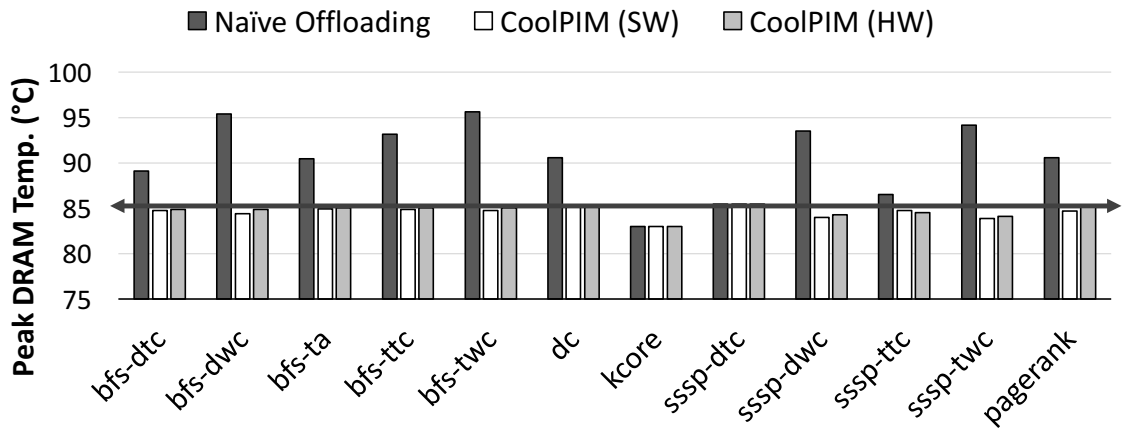


Figure 5.15: Peak DRAM temperature

Figure 5.15 shows the thermal evaluation results. In the results, with the naïve-offloading method, the peak DRAM temperature exceeds $90\,^{\circ}\mathrm{C}$ for most benchmarks. Some of them, such as `bfs-dwc` and `bfs-twc`, even reach $95\,^{\circ}\mathrm{C}$. However, with our proposed CoolPIM, all benchmarks maintain below $85\,^{\circ}\mathrm{C}$, keeping the HMC cube at a cool state.

Software-based vs. Hardware-based

As explained in Section 5.3.4, the software-based dynamic throttling usually introduces much longer throttling delay than the hardware-based method. However, because of the long thermal response time, such difference in throttling delay may not be significant in the overall control delay. To evaluate its impact, we also analyze the PIM rate over time

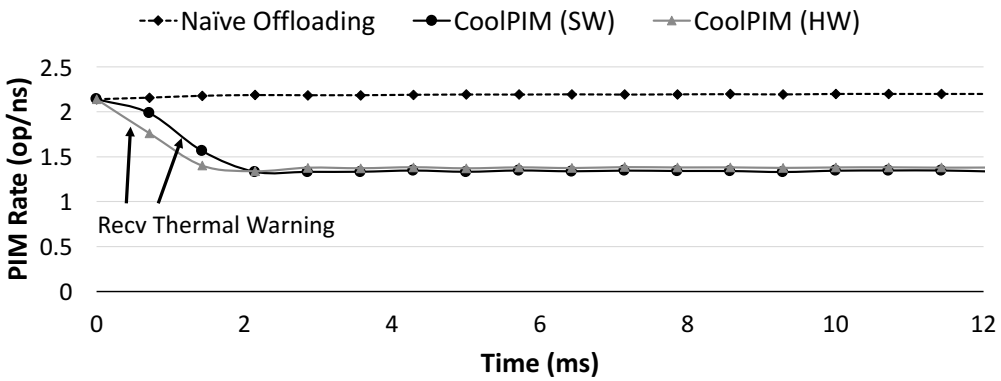


Figure 5.16: Illustration of PIM rate variation over time

during our experiments. We sample the target benchmarks at the granularity of one millisecond and compare the PIM rate variations of software- and hardware-based methods. In our results, we observe only sub-millisecond difference in the overall control delay. To better illustrate the observation, Figure 5.16 shows the PIM rate variations over time for bfs-ta benchmark. We select this benchmark because of its relatively larger delay difference and longer execution time. As shown in Figure 5.16, naïve offloading maintains extremely high PIM offloading rate with only small variations. However, both software- and hardware-based CoolPIM methods successfully control the PIM rate and keep it within a proper range. Although software-based method consumes close-to one more millisecond than hardware-based one, it still takes only trivial time compared to the millisecond-level thermal response time and the long total execution time.

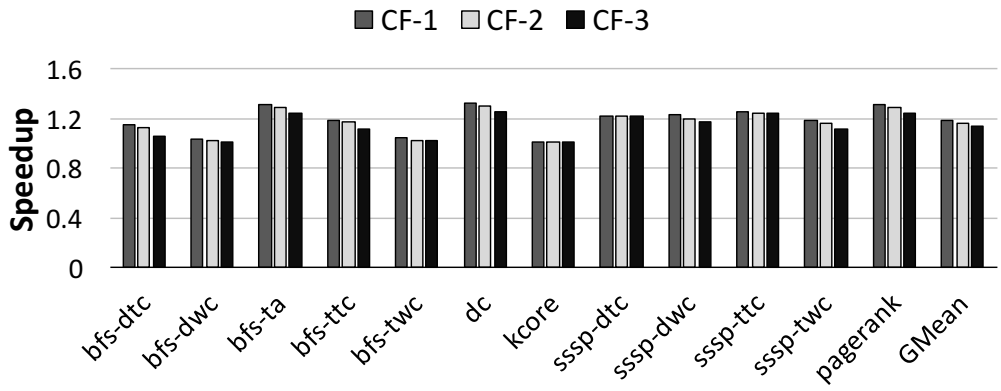


Figure 5.17: Sensitivity to control factor

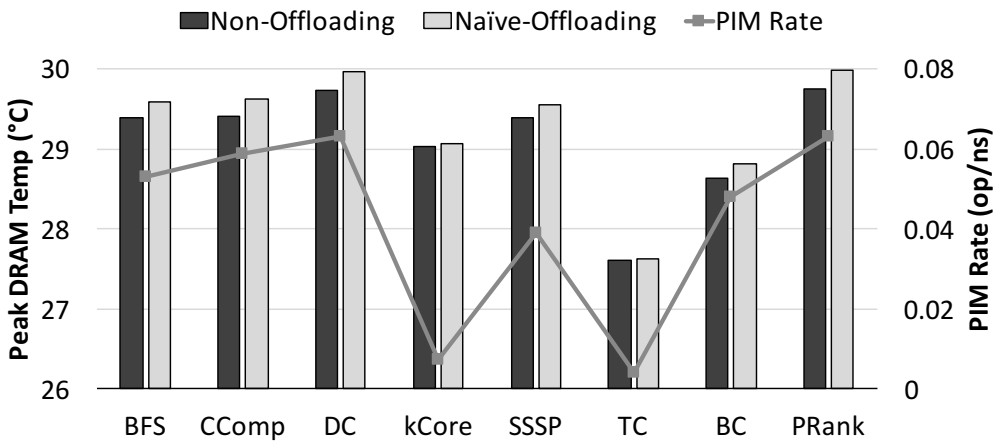


Figure 5.18: Peak DRAM temperature of CPU workloads

Parameter Sensitivity

As explained in Section 5.3.4, *Control Factor* (CF) in SW-DynT affects the reaction speed of the feedback control and the control accuracy. A long reaction time may keep the HMC cube at a hot state for longer time, and a over-reduced PTP value may lose the performance potential of a higher PIM intensity. In this section, we evaluate the sensitivity to the CF values. Figure 5.17 shows the speedup over the non-offloading system with different CF values. As shown, changing from CF-1 to CF-3 only shows a negligible performance degradation. In the worst case, `bfs-dtc` shows a 10% reduction in performance between CF-1 and CF-3. On average, the speedup variation is around 5%. The result shows a general insensitive behavior of the CF settings. This is because the difference between various CF values is relatively small compare to the large number of SMs and CUDA blocks. In addition, our proposed method uses a static mechanism to select the proper initial state. It ensures a small distance between the initial state and the desirable configuration. Thus, the reaction time would be generally short regardless of the CF variation.

Thermal Analysis of CPU Graph Workloads

As explained in prior sections, because of the high bandwidth utilization and frequent PIM offloading, graph workloads on GPUs may trigger the thermal issue and suffer from the

caused memory performance degradation. However, the graph workloads on CPUs usually do not have the thermal issue because of its low bandwidth utilization and PIM operation rate. To demonstrate that, we perform a thermal evaluation on CPU graph workloads from the GraphBIG benchmark suite following the same configuration as in Section 4.4. As shown in Figure 5.18, the peak DRAM temperatures of all workloads are below $30\,^{\circ}\mathrm{C}$ even with naïve offloading because of the low bandwidth utilization of CPU workloads. Moreover, the temperature differences between non-offloading and naïve offloading are all below $1\,^{\circ}\mathrm{C}$. This is because the extremely low PIM operation rates, which are all lower than 0.07 operation/ns, bring only negligible thermal impact. Therefore, from the results, we can observe that because of the low bandwidth utilization and PIM operation rate, graph workloads on CPU-based platforms do not trigger thermal issues, and thus do not require a thermal-aware offloading management mechanism.

5.5 Summary

Processing-in-memory (PIM) offers promising performance and energy gains, but its thermal constraints can prevent applications from benefiting its full potential. To understand the thermal impact of PIM, this dissertation performs an analysis on a prototype of HMC across various bandwidth utilization and cooling solutions. Our results show that naïvely using PIM offloading functionality causes a thermal bottleneck and degrades system performance even compared to the non-offloading case depending on the workloads. Based on our findings, we propose CoolPIM, a source throttling technique that controls PIM offloading intensity to keep the operating temperature in check. CoolPIM presents both hardware and software mechanisms to overcome the thermal bottleneck in HMC. The hardware method provides fast feedback reaction and fine-grained control, while the software method requires only non-intrusive changes in software runtime and compiler. Compared to the non-offloading and naïve offloading systems, CoolPIM improves performance up to 40% and 37% for a set of graph workloads by effectively managing thermal constraints.

CHAPTER 6

CONCLUSION

Graph computing has been widely used in a variety of domains. However, because of the inefficiency in memory subsystems, graph computing does not perform well on conventional architectures, and thus require architectural innovations for improving its execution efficiency. This dissertation addresses the bottlenecks of graph computing by utilzing the processing-in-memory (PIM) technique, which can improve memory subsystem efficiency by offloading operations on the irregular data to the memory side. This dissertation first analyzed the behavior of graph computing by proposing benchmarking mechanisms for modern graph computing systems. Based on the characterization observations, it then exploited enabling PIM offloading for graph computing on multiple hardware platforms. By merging NDP concept and graph computing together, this dissertation starts from the key question that how to enable PIM offloading for improving graph computing efficiency on CPU- and GPU-based architectures. On CPU-based platforms, it identified the key bottlenecks and proposed a full-stack solution for utilizing PIM functionality in graph frameworks. On GPU-based platforms, this dissertation showed that thermal constraints become a key obstacle because of the high memory bandwidth and PIM unit utilization, and then proposed source throttling mechanisms for thermal-aware PIM offloading. The proposed techniques in this dissertation are further summarized as follows.

- Chapter 3 presented a suite of CPU/GPU graph benchmarks. The proposed benchmark suite addressed all key benchmarking factors simultaneously by utilizing System G framework design and following a comprehensive workload selection procedure. From the characterization, it observed: 1) Conventional architectures do not perform well for graph computing. Significant inefficiencies are observed in CPU memory subsystems and GPU warp/memory bandwidth utilizations. 2) Significant

diverse behaviors are shown in different workloads and different computation types. Such diversity exists on both CPU and GPU sides, and involves multiple architectural features. 3) Graph computing on both of CPUs and GPUs are highly data sensitive. Input data has significant and complex impacts on multiple architecture features.

- Chapter 4 proposed GraphPIM, a full-stack solution that enables PIM instruction offloading for graph computing. GraphPIM is built on the key observation that the atomic access to the graph property is the main culprit for the inefficient execution of graph workloads on modern computing systems. Thus, by offloading the atomic operations on the graph property to the PIM side, GraphPIM avoids the overhead of executing atomic instructions in the host processor as well as the inefficient utilization of the memory subsystem caused by irregular data accesses. GraphPIM achieves up to a 2.4× speedup and a reduction of 37% in energy consumption for a wide range of graph benchmarks and real-world applications.

- Chapter 5 offers an analysis on a prototype HMC system across a wide range of bandwidth utilization and cooling solutions. Because of the thermal constraints, naïvely using PIM offloading functionality can cause a thermal issue and degrades system performance even compared to the non-offloading case depending on the workloads. Hence, this chapter proposed CoolPIM, a source throttling technique that controls PIM offloading intensity to keep the operating temperature in check. CoolPIM improves performance up to 40% and 37% for a set of graph workloads compared to the non-offloading and naïve offloading systems by managing thermal constraints effectively.

The emergence of data science and the increment of data volumes require more efficient processing of large-scale graphs. This dissertation provides a set of architectural solutions for improving graph computing efficiency. The proposed solutions are based on the real-world NDP standard from HMC 2.0 and the comprehensive understanding of real-

world graph computing systems. The solutions covers both CPU- and GPU-based host architectures with considerations of software system design as well as hardware thermal constraints. Requiring only minor non-intrusive modifications of software and hardware, the proposed solutions can serve as an effective and practical starting point for future graph computing architecture designs.

REFERENCES

[1] C.-Y. Lin, L. Wu, Z. Wen, H. Tong, V. Griffiths-Fisher, L. Shi, and D. Lubensky, "Social network analysis in enterprise," *Proceedings of the IEEE*, vol. 100, no. 9, 2012.

[2] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, "Information network or social network?: The structure of the twitter follow graph," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW Companion'14, Seoul, Korea, 2014, ISBN: 978-1-4503-2745-9.

[3] Z. Wen and C.-Y. Lin, "How accurately can one's interests be inferred from friends," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW'10, Raleigh, North Carolina, USA: ACM, 2010, ISBN: 978-1-60558-799-8.

[4] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, 2003.

[5] D. Bae, K. Han, J. Park, and M. Y. Yi, "Apptrends: A graph-based mobile app recommendation system using usage history," in *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, IEEE, 2015, pp. 210–216.

[6] T. Aittokallio and B. Schwikowski, "Graph-based methods for analysing networks in cell biology," *Briefings in bioinformatics*, vol. 7, no. 3, pp. 243–255, 2006.

[7] J. Degac, U. Winter, and V. Helms, "Graph-based clustering of predicted ligand-binding pockets on protein surfaces," *Journal of chemical information and modeling*, vol. 55, no. 9, pp. 1944–1952, 2015.

[8] M. Canim and Y.-C. Chang, "System G Data Store: Big, rich graph data analytics in the cloud," in *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, ser. IC2E'13.

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," ser. SIGMOD'10, Indianapolis, Indiana, USA, 2010, ISBN: 978-1-4503-0032-2.

[10] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, Hollywood, CA, 2012, ISBN: 978-1-931971-96-6.

[11] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM'09, Washington, DC, USA, 2009, ISBN: 978-0-7695-3895-2.

[12] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric Graph Processing on GPUs," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC'14, Vancouver, BC, Canada, 2014.

[13] I. Tanase, Y. Xia, L. Nai, Y. Liu, W. Tan, J. Crawford, and C.-Y. Lin, "A highly efficient runtime and graph library for large scale graph analytics," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES'14, Snowbird, UT, USA.

[14] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'13, New York, New York, USA, 2013, ISBN: 978-1-4503-2037-5.

[15] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW'13, Rio de Janeiro, Brazil, 2013, ISBN: 978-1-4503-2035-1.

[16] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP'13, Farminton, Pennsylvania, 2013, ISBN: 978-1-4503-2388-8.

[17] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, Apr. 2012.

[18] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14, Broomfield, CO: USENIX Association, 2014, pp. 599–613, ISBN: 978-1-931971-16-4.

[19] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "GraphReduce: Processing large-scale graphs on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, Austin, Texas: ACM, 2015, 28:1–28:12, ISBN: 978-1-4503-3723-6.

[20] L. Nai, Y. Xia, C.-Y. Lin, B. Hong, and H.-H. S. Lee, "Cache-conscious graph collaborative filtering on multi-socket multicore systems," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF'14, Cagliari, Italy: ACM, 2014, ISBN: 978-1-4503-2870-8.

[21] Y. Xia, J.-H. Lai, L. Nai, and C.-Y. Lin, "Concurrent image query using local random walk with restart on large scale graphs," in *2014 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, ser. ICMEW'14, 2014.

[22] Y. Xia and V. K. Prasanna, "Topologically adaptive parallel breadth-first search on multicore processors," in *Proceedings of 21st International Conference on Parallel and Distributed Computing Systems*, ser. PDCS'09, 2009.

[23] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2013, pp. 463–474.

[24] M. Gokhale, D. S. Res., B. Holmes, and K. Iobst, "Processing in memory: the terasys massively parallel pim array," in *IEEE Computer*, vol. 28, IEEE, 1995, pp. 23–31.

[25] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Back to Results Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *Supercomputing, ACM/IEEE 1999 Conference*, IEEE, 1999, pp. 23–31, ISBN: 1-58113-091-0.

[26] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *Proceedings of international Conference on Computer Design, 1999. (ICCD '99)*, 1999, pp. 192–201.

[27] M. Oskin, F. Chong, and T. Sherwood, "Active pages: a computation model for intelligent memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, IEEE, 1998, pp. 192–203, ISBN: 0-8186-8491-7.

[28] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick, "Intelligent RAM (IRAM): The Industrial Setting, Applications, and Architectures," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, IEEE, 1997, pp. 2–7, ISBN: 0-8186-8206-X.

[29] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proceeding of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[30] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, Oregon: ACM, 2015, pp. 105–117, ISBN: 978-1-4503-3402-0.

[31] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, Oregon: ACM, 2015, pp. 336–348, ISBN: 978-1-4503-3402-0.

[32] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *VLSI Technology (VLSIT), 2012 Symposium on*, 2012, pp. 87–88.

[33] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips 23*, 2011.

[34] Apache, *Apache Giraph*, http://giraph.apache.org/, 2014.

[35] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi, "PGX.D: A Fast Distributed Graph Processing Engine," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, Austin, Texas: ACM, 2015, 58:1–58:12, ISBN: 978-1-4503-3723-6.

[36] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, London, England, UK: ACM, 2012.

[37] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," in *Cray User's Group (CUG)*, 2010.

[38] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA'14.

[39] D. A. Bader and K. Madduri, *Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors*, 2005.

[40] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in*

Algorithms and Architectures, ser. SPAA'12, Pittsburgh, Pennsylvania, USA, 2012, ISBN: 978-1-4503-1213-4.

[41] The IMPACT Research Group, UIUC, *Parboil benchmark suite*, `http://impact.crhc.illinois.edu/parboil.php`.

[42] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[43] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 65–76.

[44] Standard Performance Evaluation Corporation, *Spec cpu 2006*, `http://www.spec.org/cpu2006`.

[45] J. Leskovec and A. Krevl, *Snap datasets: Stanford large network dataset collection*, Jun. 2014.

[46] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "Benchmarking graph-processing platforms: A vision," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE'14, Dublin, Ireland, 2014.

[47] A. L. Varbanescu, M. Verstraaten, C. de Laat, A. Penders, A. Iosup, and H. Sips, "Can portability improve performance?: An empirical study of parallel graph analytics," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE'15, Austin, Texas, USA, 2015, ISBN: 978-1-4503-3248-4.

[48] D. Elliott, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational ram: implementing processors in memory," *Design Test of Computers, IEEE*, vol. 16, no. 1, pp. 32–41, 1999.

[49] Y. Kim, T.-D. Han, S.-D. Kim, and S.-B. Yang, "An effective memory-processor integrated architecture for computer vision," in *Proceedings of the 1997 International Conference on Parallel Processing, 1997.*, 1997, pp. 266–269.

[50] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 a 1.2v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods

using 29nm process and tsv," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, 2014, pp. 432–433.

[51] R. Nair, S. Antao, C. Bertolli, P. Bose, J. Brunheroto, T. Chen, C. Cher, C. Costa, J. Doi, C. Evangelinos, B. Fleischer, T. Fox, D. Gallo, L. Grinberg, J. Gunnels, A. Jacob, P. Jacob, H. Jacobson, T. Karkhanis, C. Kim, J. Moreno, J. O'Brien, M. Ohmacht, Y. Park, D. Prener, B. Rosenburg, K. Ryu, O. Sallenave, M. Serrano, P. Siegl, K. Sugavanam, and Z. Sura, "Active memory cube: a processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, 17:1–17:14, 2015.

[52] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): enabling programmer-transparent near-data processing in gpu systems," in *Proceedings of the 43nd Annual International Symposium on Computer Architecture*, ser. ISCA '16, 2016.

[53] G. H., L. N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A processing-in-memory taxonomy and a case for studying fixed-function pim," in *WoNDP: 1st Workshop on Near-Data Processing*, MICRO 46, 2013.

[54] L. Nai and H. Kim, "Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15, Washington DC, DC, USA: ACM, 2015, pp. 258–261, ISBN: 978-1-4503-3604-8.

[55] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3D Systems Integration Conference (3DIC), 2013 IEEE International*, 2013, pp. 1–7.

[56] L. Xu, D. P. Zhang, N. Jayasena, D. Zhang, A. Farmahini-Farahani, M. Ignatowski, A. Lyashevsky, and J. Greathouse, "Scaling deep learning on multiple in-memory processors," in *3rd Workshop on Near-Data Processing In conjunction with MICRO-48*, ser. WoNDP'03, 2015.

[57] H. M. C. Consortium, "Hybrid Memory Cube Specification 2.0," *Retrieved from hybridmemorycube.org*, 2014.

[58] D. Milojevic, S. Idgunji, D. Jevdjic, E. Ozer, P. Lotfi-Kamran, A. Panteli, A. Prodromou, C. Nicopoulos, D. Hardy, B. Falsari, and Y. Sazeides, "Thermal characterization of cloud workloads on a power-efficient server-on-chip," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, 2012, pp. 175–182.

[59] JEDEC, *Wide I/O Single Data Rate (Wide I/O SDR)*, 2011.

[60] Y. Eckert, N. Jayasena, and G. H. Loh, "Thermal feasibility of die-stacked processing in memory," in *The 2nd Workshop on Near-Data Processing (WoNDP)*, 2014.

[61] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *ACM SIGARCH computer architecture news*, IEEE Computer Society, vol. 36, 2008, pp. 453–464.

[62] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: a programmable digital neuromorphic architecture with high-density 3d memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 380–392.

[63] M. Guan and L. Wang, "Improving DRAM Performance in 3D ICs via Temperature Aware Refresh," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, no. 99, pp. 1–11, 2016.

[64] Micron, *4gb: x4, x8, x16 ddr4 sdram features*, `https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/4gb_ddr4_sdram.pdf`.

[65] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 1–12.

[66] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 489–501.

[67] *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman, 2002.

[68] J. Webber, "A programmatic introduction to neo4j," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12, 2012.

[69] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Proceedings of the 14th international conference on High performance computing*, ser. IISWC'12, 2012.

[70] Y. Xia, "System G: Graph Analytics, Storage and Runtimes," in *Tutorial on the 19th PPoPP*, 2014.

[71] R. W. Williams and K. M. Megan, "Genetic and molecular network analysis of behavior," *Int Rev Neurobiol*, 2012.

[72] Y. Bengio, "Learning deep architectures for ai," *Found. Trends Mach. Learn.*, vol. 2, no. 1, Jan. 2009.

[73] E. Chesler and M. Haendel, *Bioinformatics of Behavior:* pt. 2. Elsevier Science, 2012, ISBN: 9780123983107.

[74] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, Jul. 1983.

[75] J. Soman, K. Kishore, and P. Narayanan, "A fast GPU algorithm for graph connectivity," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, ser. IPDPSW'10.

[76] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*.

[77] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM J. Sci. Comput.*, May 1993.

[78] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *2009 IEEE International Symposium on Parallel Distributed Processing*, ser. IPDPS'09, 2009.

[79] U. Kang, S. Papadimitriou, J. Sun, and H. Tong, "Centralities in large networks: Algorithms and observations," in *Proceedings of the 2011 SIAM International Conference on Data Mining*, ser. SDM'11, 2011.

[80] M.-D. Pham and et al., "S3G2: A Scalable Structure-Correlated Social Graph Generator," in *Selected Topics in Performance Evaluation and Benchmarking: 4th TPC Technology Conference, TPCTC 2012, Istanbul, Turkey, August 27, 2012, Revised Selected Papers*, ser. TPCTC'12.

[81] S. Andreassen, F. V. Jensen, S. K. Andersen, B. Falck, U. Kjærulff, M. Woldbye, A. R. Sørensen, A. Rosenfalck, and F. Jensen, "MUNIN — an expert EMG assistant," in *Computer-Aided Electromyography and Expert Systems*, 1989.

[82] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 445–456.

[83] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*. San Francisco, CA, USA, 2015.

[84] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014, pp. 35–44.

[85] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 78–88, ISBN: 978-0-7695-4566-0.

[86] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[87] S. Lee, T. Johnson, and E. Raman, "Feedback directed optimization of tcmalloc," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14, Edinburgh, United Kingdom: ACM, 2014, 3:1–3:8, ISBN: 978-1-4503-2917-0.

[88] *Intel a64 and ia-32 architectures software developer's manual: Volume 3a:system programming guide*, Intel Corporation, 2015.

[89] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions," in *Proceedings of the 2015 ACM/IEEE conference on Supercomputing*, ACM, 2015.

[90] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, Mar. 2011.

[91] *Macsim*, http://code.google.com/p/macsim/.

[92] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, 2011.

[93] P. Rosenfeld, "Performance exploration of the hybrid memory cube," Ph.D. dissertation, University of Maryland, College Park, 2014.

[94] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 145–155.

[95] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, "The LDBC Social Network Benchmark: Interactive Workload," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: ACM, 2015, pp. 619–630, ISBN: 978-1-4503-2758-9.

[96] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, *Cacti 6.0: a tool to understand large caches*, 2009.

[97] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, New York, New York: ACM, 2009, pp. 469–480, ISBN: 978-1-60558-798-1.

[98] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014, pp. 190–200.

[99] G. Sadowksi and P. Rathle, "Fraud detection: Discovering connections with graph databases," Neo Technology, Inc., Tech. Rep., 2015.

[100] D. Ron and A. Shamir, "Financial cryptography and data security: 17th international conference, fc 2013, okinawa, japan, april 1-5, 2013, revised selected papers," in, A.-R. Sadeghi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. Quantitative Analysis of the Full Bitcoin Transaction Graph, pp. 6–24, ISBN: 978-3-642-39884-1.

[101] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th International Conference on World Wide Web*, ser. WWW '01, Hong Kong, Hong Kong: ACM, 2001, pp. 285–295, ISBN: 1-58113-348-0.

[102] R. Zafarani and H. Liu, "Social computing data repository at ASU," *School of Computing, Informatics and Decision Systems Engineering, Arizona State University*, 2009.

[103] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 445–456.

[104] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: enabling instruction-level PIM offloading in graph computing frameworks," in *Proceedings*

of the IEEE 23th International Symposium on High Performance Computer Architecture (HPCA), 2017.

[105] M. Elteir, H. Lin, and W.-c. Feng, "Performance characterization and optimization of atomic operations on AMD GPUs," in *2011 IEEE International Conference on Cluster Computing*, 2011, pp. 234–243.

[106] D. Patterson, "The top 10 innovations in the new NVIDIA fermi architecture, and the top 3 next challenges," *NVIDIA Whitepaper*, vol. 47, 2009.

[107] Samsung, *The future of graphic and mobile memory for new applications*, Hot Chips 28 Tutorial.

[108] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The Architecture of the DIVA Processing-in-memory Chip," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02, 2002, pp. 14–25.

[109] JEDEC, *Ddr4 sdram specification*, 2013.

[110] NVIDIA Corporation, *Gp100 pascal whitepaper*, `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

[111] Advanced Micro Devices Inc., *Radeon r9 fury x*, `http://support.amd.com/Documents/amd-radeon-r9-fury-x.pdf`.

[112] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[113] Picocomputing, *Sc6-mini*, `http://picocomputing.com/products/picocube/picomini/`, [Online; accessed 19-November-2016].

[114] Picocomputing, *Ex700 backplane*, `http://picocomputing.com/products/backplanes/ex-700/`, [Online; accessed 19-November-2016].

[115] Picocomputing, *Ac-510 hpc module*, `http://picocomputing.com/ac-510-superprocessor-module/`, [Online; accessed 19-November-2016].

[116] HMC Consortium, "Hybrid Memory Cube Specification 1.0," *Retrieved from hybridmemorycube.org*, 2013.

[117] L. P. Eric Bogatin Dick Potter, *Roadmaps of Packaging Technology*. Integrated Circuit Engineering Corporation, 1997, pp. 6–1–6–32, ISBN: 1-877750-61-1.

[118] W. J. Song, S. Mukhopadhyay, and S. Yalamanchili, "Kitfox: multi-physics libraries for integrated power, thermal, and reliability simulations of multicore microarchitecture," *IEEE Transactions on Component, Packaging, and Manufacturing Technology*, vol. 5, no. 11, pp. 1590–1601, 2015.

[119] A. Sridhar, A. Vincenzi, M. Ruggiero, T. Brunschwiler, and D. Atienza, "3D-ICE: Fast Compact Transient Thermal Modeling for 3D ICs with Inter-Tier Liquid Cooling," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2010, pp. 463–470.

[120] J. Stein and M. M. Hydeman, "Development and testing of the characteristic curve fan model," *ASHRAE Transactions*, vol. 110, no. 1, pp. 347 –356, 2004.

[121] P. Teertstra, M. M. Yovanovich, J. R. Culham, and T. Lemczyk, "Analytical forced convection modeling of plate fin heat sinks," in *Semiconductor Thermal Measurement and Management Symposium, 1999. Fifteenth Annual IEEE*, 1999, pp. 34–41.

[122] synopsys, *Synopsys 32/28nm generic library*, `https://www.synopsys.com/COMMUNITY/UNIVERSITYPROGRAM/Pages/32-28nm-generic-library.aspx`, 2016.

[123] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "High Performance AXI-4.0 Based Interconnect for Extensible Smart Memory Cubes," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 1317–1322.

[124] N. B Lakshminarayana, "Efficient graph algorithm execution on data-parallel architectures," PhD thesis, Georgia Institute of Technology, 2014.